Programmierung des
6502

RODNAY ZAKS



programmierung des 6502

6502 Serie – Band I

programmierung des 6502

RODNAY ZAKS

2. überarbeitete Ausgabe

ANMERKUNGEN

Der Verlag hat alle Sorgfalt walten lassen, um vollständige und akkurate Informationen zu publizieren. SYBEX-Verlag GmbH. übernimmt keine Verantwortung für die Nutzung dieser Informationen, auch nicht für die Verletzung von Patent-, Lizenz- und anderen Rechten Dritter, die daraus resultieren. Es ist keine Lizenz von Herstellern erteilt worden und es sei insbesondere darauf hingewiesen, daß Hersteller ihre Schaltpläne ändern, ohne die breite Öffentlichkeit davon zu unterrichten. Technische Charakteristika und Preise können einem rapidem Wechsel ausgesetzt sein. Für die neuesten technischen Daten ist es daher empfohlen, die Angaben der Hersteller zur Hand zu nehmen.

Originalausgabe in Englisch
Titel der englischen Ausgabe: PROGRAMMING THE 6502
Original Copyright © 1980 SYBEX Inc., Berkeley, California, USA.

Deutsche Übersetzung: Bernd Pol

Umschlag: Daniel Le Noury
Gesamtherstellung: Hub. Hoch, Düsseldorf

ISBN 3-88745-011-6
1. Auflage 1980
2. Auflage 1981
3. Auflage 1982

Vorwort

Dieses Buch ist als vollständiger und in sich abgeschlossener Text zum Erlernen des Programmierens mit dem 6502-Prozessor erstellt worden. Es kann ohne vorherige Programmierkenntnisse benutzt werden, sollte aber auch allen anderen 6502-Programmierern etwas bringen.
Jeder mit Programmiererfahrung kann dem Buch die, den besonderen 6502-Eigenschaften entsprechenden Programmiertechniken entnehmen (und lernen, wie man die 6502-typischen Einschränkungen umgeht). Mit dem Text werden die elementaren Techniken zur effektiven Programmerstellung bis hin zu mittleren Anforderungen abgedeckt.
Der Text hat zum Ziel, jedem, der das Programmieren mit diesem Mikroprozessor zu erlernen wünscht, alle notwendigen Kenntnisse zu dessen Beherrschung zu vermitteln. Natürlich kann kein Buch alleine das Programmieren lehren, wenn nicht die notwendige Praxis dazukommt. Es ist jedoch zu hoffen, daß der Leser sich soweit führen läßt, bis er die Überzeugung gewinnt, selbst mit dem Programmieren beginnen zu können, um so einfache oder sogar relativ komplexe Probleme mit Hilfe eines Mikrocomputers zu lösen.
Die Arbeit beruht auf der Unterrichtserfahrung des Autors, der mehr als 1.000 Interessenten das Programmieren von Mikrocomputern gelehrt hat. Im Ergebnis dieser Erfahrung ist das Buch streng gegliedert worden. Jedes Kapitel geht normalerweise vom Einfachen zum Komplizierten vor. Leser, die bereits elementare Programmierkenntnisse haben, können das erste Kapitel überspringen. Andere ohne solche Erfahrung werden die abschließenden Abschnitte einiger Kapitel mehrfach durcharbeiten müssen. Dabei ist das Buch so angelegt, daß der Leser systematisch alle grundlegenden Konzepte der Techniken zur Erstellung von immer komplexeren Programmen erlernen kann. Es wird daher nahegelegt, die Reihenfolge des Dargestellten einzuhalten. Für einen ausreichenden Lernerfolg ist es weiter wichtig, daß der Leser versucht, so viele Übungsaufgaben wie nur möglich zu lösen. Der Schwierigkeitsgrad dieser Übungen ist sorgfältig abgewogen worden. Sie sind so entworfen, daß sich mit ihnen das Verständnis des Textes überprüfen läßt. Ohne diese Übungsaufgaben zu lösen ist es nicht möglich, den Gehalt des Buchs als Lehrmaterial voll auszuschöpfen. Einige dieser Übungen benötigen relativ viel Zeit zur Lösung, wie beispielsweise die Multiplikationsübung im dritten Kapitel. Wenn man diese Zeit jedoch investiert, so wird man tatsächlich programmieren und dadurch *durch die eigene Arbeit lernen* können. Und das ist eigentlich unabdingbar.

Wer beim Durcharbeiten des vorliegenden Buchs auf den Geschmack am Programmieren gekommen sein sollte, kann zu einem weiterführenden Werk greifen, dem „6502 Applications Book“, das in Kürze auch in einer deutschen Ausgabe vorliegen wird.
Andere Bücher der SYBEX-Reihe beschäftigen sich mit dem Programmieren anderer weitverbreiteter Mikroprozessoren.
Zur Lösung von Hardwarefragen sind die Bücher „FROM CHIPS TO SYSTEMS: An Introduction to Microprocessing“, das auch bald in einer deutschen Ausgabe erscheinen wird, und „MIKROPROZESSOR INTERFACE TECHNIKEN“ empfohlen.
Der Inhalt des vorliegenden Buches ist sorgfältig überprüft worden. Es ist jedoch nicht zu vermeiden, daß hier und da noch Druck- und andere Fehler vorliegen. Der Autor ist für jeden Leserhinweis dankbar, um so in kommenden Auflagen diese Erfahrung einfließen lassen zu können. Dementsprechend sind alle weiteren Anregungen, das Buch zu verbessern, wie z. B. weitere Programmbeispiele, sei es als Wunsch oder als vom Leser ausgearbeitetes und für wertvoll erachtetes Programm, herzlich willkommen.

Vorwort zur 2. überarbeiteten Ausgabe

Die zweite Auflage umfaßte nahezu 100 Seiten mehr als die erste, wobei der größte Teil des Materials in Kapitel 1 und Kapitel 9, d. h. an den beiden Enden des Spektrums eingebracht worden ist. Kapitel 1 ist die allgemeine Einführung. Kapitel 9 enthält weitergehende Informationen über Datenstrukturen.
Weitere Verbesserungen sind im ganzen Buch vorgenommen worden, und ich möchte den vielen Lesern der vorigen Auflage danken, die wertvolle Verbesserungsanregungen gemacht hatten.
Besonders seien hier die Anregungen von Eric Martinot und Chris Williams für die komplexeren Programmierbeispiele in Kapitel 9 zu erwähnen, ebenso Daniel J. David wegen der vielen Verbesserungsvorschläge, die er einbrachte. Eine Großzahl weiterer Änderungen und Verbesserungen gehen auf die wertvollen Untersuchungen und Anmerkungen von Philip K. Hooper, John Smith, Roland Long, Charles Curley, N. Harris, John McClenon, Douglas Trusty und Fletcher Carson zurück.
Die 2. überarbeitete Ausgabe ist von Herrn Prof. Dr. Lutz Richter redaktionell überarbeitet worden. Alle in den Abbildungen enthaltenen Textteile, die in den früheren Ausgaben noch im englischen erschienen, sind ins deutsche übertragen worden. Außerdem haben wir alle Abbildungen überarbeiten lassen, um somit nicht nur die drucktechnische Wiedergabe zu verbessern, sondern auch um eine maximale Informationswiedergabe zu gewährleisten.

Die 6502 Reihe:

Band I –	*PROGRAMMIERUNG DES 6502*
Band II –	*6502 ANWENDUNGSBUCH*
Band III –	*6502 ADVANCED APPLICATIONS*

SOFTWARE:	*6502 ASSEMBLER IN BASIC*
	8080 SIMULATOR FOR APPLE
	8080 SIMULATOR FOR KIM
	6502 GAMES
	6502 APPLICATIONS

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

KAPITEL 1
GRUNDLAGEN

Einführung

In diesem Kapitel sollen die grundlegenden Konzepte und Definitionen eingeführt werden, auf denen das Programmieren eines Computers beruht. Der mit diesen Konzepten bereits vertraute Leser mag den Inhalt des Kapitels rasch überfliegen und dann mit Kapitel 2 weitermachen. Es ist jedoch auch dem erfahrenen Leser empfohlen, sich dieses Einführungskapitel wenigstens kurz anzusehen. Es werden hier viele wichtige Begriffe eingeführt, wie z. B. das Zweierkomplement, BCD-Zahlen und andere Zahlendarstellungen. Einige dieser Konzepte mögen dem Leser neu sein, andere dürften die Kenntnisse und Fertigkeiten des erfahreneren Programmierers erweitern helfen.

Was ist Programmieren?

Wenn man eine Aufgabe gestellt bekommt, so muß man zunächst einen Lösungsweg entwerfen. Dieser Lösungsweg, dargestellt als schrittweises Vorgehen, wird *Algorithmus* genannt. Ein Algorithmus ist eine Schritt für Schritt vorgehende Ausarbeitung des Lösungswegs für ein vorgegebenes Problem. Er muß in einer endlichen Anzahl von Schritten zu einem Ende führen. Dieser Algorithmus kann in jeder beliebigen Sprache, in beliebigen Symbolismen ausgedrückt werden. Ein einfaches Beispiel für einen Algorithmus ist folgendes:

1 – Schlüssel ins Schloß stecken
2 – Schlüssel einmal ganz links herum drehen
3 – Türgriff erfassen,
4 – Türgriff nach links drehen und Tür schieben.

Wenn der Algorithmus für das betrachtete Schloß zutrifft, so sollte an dieser Stelle die Tür offen sein. So stellt sich diese Vierschrittprozedur als Algorithmus zum Öffnen einer Tür dar.

Ist der Weg zur Lösung eines Problems erst einmal in Form eines Algorithmus festgehalten, so muß dieser Algorithmus vom Computer ausgeführt werden. Leider steht es unverrückbar fest, daß zur Zeit kein Computer die deutsche oder sonst eine Umgangssprache versteht. Der Grund liegt an der *syntaktischen Mehrdeutigkeit* aller menschlichen Umgangssprachen. Ein Computer kann nur eine wohldefinierte Untermenge der natürlichen Sprache „verstehen“. Man nennt diese eine *Programmiersprache.*

Das Übertragen eines Algorithmus in eine Folge von Befehlen einer Programmiersprache nennt man *Programmieren.* Genauer gesagt, bezeichnet man die Phase, in

der der Algorithmus in die Programmiersprache übersetzt wird, als *Kodierung*. Programmieren bezieht sich nicht auf das Kodieren alleine, sondern umfaßt die gesamte Entwicklung des Programms und die „Datenstrukturen", mit denen der Algorithmus in die Praxis umgesetzt, *implementiert* wird.
Ein effektives Programmieren erfordert nicht alleine ein Verständnis der für Standardalgorithmen möglichen Programmiertechniken, sondern auch das geschickte Ausnutzen aller Hardwaremöglichkeiten des Computers, wie die internen Register, den Speicher und die peripheren Einheiten, und dazu noch den kreativen Gebrauch passender Datenstrukturen. Diese Techniken werden in den nächsten Kapiteln betrachtet werden.
Programmieren erfordert ebenfalls eine strenge Disziplin bei der Dokumentation, durch welche die Programme sowohl für andere als auch für den Autor selbst verständlich werden. Eine solche Dokumentation muß sowohl im Programm selbst als auch außerhalb davon vorliegen.
Interne Programmdokumentation umfaßt die in den Programmtext eingefügten Kommentare, mit denen seine Arbeitsweise verdeutlicht wird.
Externe Dokumentation bezieht sich auf Entwurfsdokumente, die unabhängig vom Programm selbst vorliegen: geschriebene Erläuterungen, Handbücher und Flußdiagramme.

Flußdiagramme

Ein besonderer Schritt wird fast immer zwischen *Algorithmus* und *Programm* eingeschoben. Man bezeichnet ihn als *Flußdiagramm.* Ein Flußdiagramm ist einfach nur eine symbolische Wiedergabe des Algorithmus in eine Abfolge von Rechtecken und Rauten, die die einzelnen Schritte des Algorithmus enthalten. Rechtecke werden für *Befehle* eingesetzt. Rauten werden für *Entscheidungen* gebraucht, wie z.B.: Wenn Information X wahr ist, dann führe Aktion A aus, sonst Aktion B. Wir verzichten hier jedoch noch darauf, eine formale Definition von Flußdiagrammen an dieser Stelle einzuführen und werden ihre Einführung und Diskussion auf später verschieben, wenn Programme auszuarbeiten sind.
Das Erstellen von Flußdiagrammen ist als Schritt zwischen der Ausarbeitung des Algorithmus und der eigentlichen Kodierung unbedingt zu empfehlen. Man hat bemerkenswerterweise beobachtet, daß ungefähr 10% aller Programmierer ein Programm ohne Flußdiagramm mit Erfolg ausarbeiten können. Leider konnte auch beobachtet werden, daß sich 90% der Programmierer als zu diesen 10% gehörig empfinden! Das Ergebnis: Im Schnitt versagen 80% dieser Programme beim ersten Probelauf auf dem Computer. (Diese Prozentzahlen sind natürlich nur Näherungswerte.) Kurz gesagt sehen die meisten Neulinge auf dem Feld des Programmierens die Notwendigkeit, ein Flußdiagramm zu zeichnen, nur selten ein. Das führt in der Regel zu „unsauberen" oder fehlerhaften Programmen. Es ist sehr viel Zeit notwendig, diese Programme auszutesten und zu korrigieren (man nennt das die „Debuggingphase" nach dem englischen „debugging" – „Entwanzen"). Es ist daher in allen Fällen sehr zu empfehlen, vor der eigentlichen Kodierung erst ein Flußdiagramm zu erstellen. Man benötigt etwas zusätzliche Zeit, erhält dafür aber ein durchschaubares Programm, das schnell und richtig abgearbeitet wird.

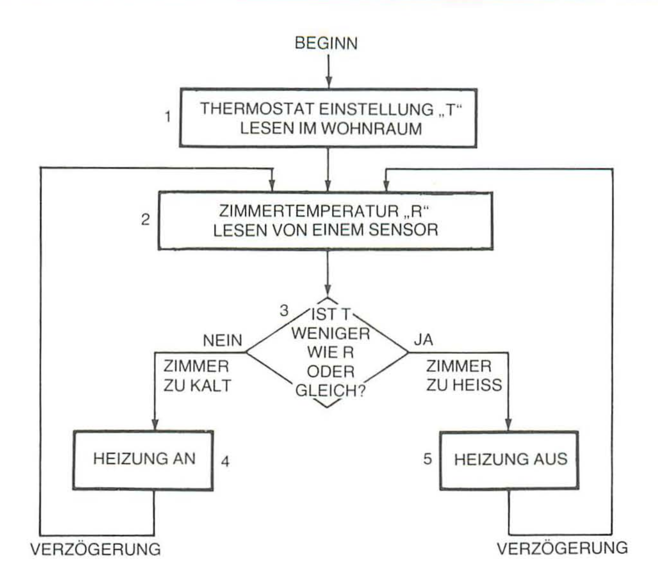


Bild 1-1: Ein Flußdiagramm zur Regelung der Zimmertemperatur

Wenn das Erstellen von Flußdiagrammen erst richtig verstanden ist, mag ein kleiner Prozentsatz von Programmierern diesen Schritt auch ohne Papier und Bleistift im Kopf ausführen können. Leider sind in solchen Fällen die entstandenen Programme in der Regel für jemand anders ohne die durch Flußdiagramme bereitgestellte Dokumentation nur schwer verständlich. So ergibt sich, daß für jedes wichtige Programm unbedingt auch die zugehörigen Flußdiagramme erstellt werden sollten. Wir werden dazu im ganzen Buch viele Beispiele finden.

Informationsdarstellung

Alle Computer gehen mit Information in Form von Zahlen oder Zeichen um. Wir wollen hier die externen und die internen Darstellungsmöglichkeiten von Information im Computer untersuchen.

Interne Informationsdarstellung

Alle Information wird in einem Computer in Form von Bitgruppen gespeichert. Das Wort *Bit* ist die Abkürzung von *binary digit* (Binärstelle, d. h. „0" oder „1"). Aufgrund der Beschränkungen der üblichen Elektronik benutzt die einzig praktikable Informationsdarstellung eine zweiwertige Logik (die Wiedergabe der Zustände „0" und „1"). Die in der Digitalelektronik verwendeten Schaltungen besitzen üblicherweise die beiden Zustände „Ein" und „Aus", und diese werden als die logischen Symbole „0" und „1" wiedergegeben. Da man derartige Schaltungen zur Implementation von „logischen" Funktionen benutzt, bezeichnet man sie mit dem Sammelnamen „Binärlogik". Im Ergebnis wird nahezu die gesamte Informationsverarbeitung heutzutage in

einem sogenannten Binärformat ausgeführt. Im allgemeinen Fall von Mikroprozessoren und speziell beim 6502 werden diese Bits in Achtergruppen zusammengefaßt. Eine Gruppe aus acht Bits nennt man ein *Byte.* Eine Gruppe von vier Bits heißt *Nibble.*

Sehen wir uns an, wie man Information intern in einem solchen Binärformat darstellt. Dabei müssen zwei Einheiten im Computer festgehalten werden. Die eine ist das Programm, d. h. die Befehlsfolge. Die andere umfaßt die Daten, mit denen das Programm arbeitet, und die Zahlen oder alphanumerischen Text umfaßt. Wir werden unten drei Darstellungsformen untersuchen: Programm, Zahlen und alphanumerischen Text.

Programmdarstellung

Alle Befehle werden intern als einzelne oder mehrfache Bytes wiedergegeben. Ein sogenannter „Kurzbefehl" benötigt nur ein einziges Byte. Ein längerer Befehl wird zur Wiedergabe zwei oder mehr Bytes brauchen. Da der 6502 ein Achtbitprozessor ist, übernimmt er die Bytes eines nach dem anderen aus seinem Speicher. Aus diesem Grund kann ein Einbytebefehl immer schneller abgearbeitet werden als ein Zwei- oder gar ein Dreibytebefehl. Wir werden später sehen, daß dies eine wichtige Eigenschaft eines jeden Mikroprozessorbefehlssatzes ist, in Sonderheit beim 6502, wo besonders viel Anstrengungen gemacht wurden, um so viele Einbytebefehle wie nur möglich bereitzustellen. Jedoch hat die Beschränkung auf eine Befehlslänge von acht Bits zu wichtigen Einschränkungen geführt, die noch dargestellt werden. Wir haben hier ein klassisches Beispiel eines Kompromisses zwischen Geschwindigkeit und Programmflexibilität vorliegen. Der zur Befehlswiedergabe benutzte Binärkode ist vom Hersteller vorgeschrieben. Der 6502 ist wie jeder andere Mikroprozessor mit einem festen Befehlssatz ausgerüstet. Diese Befehle sind vom Hersteller definiert worden und am Ende des Buchs zusammen mit ihrem Kode aufgelistet. Ein jedes Programm wird als Folge solcher binärer Befehle ausgedrückt. Die 6502-Befehle werden in allen Einzelheiten in Kapitel 4 vorgestellt.

Wiedergabe numerischer Daten

Es ist nicht ganz so einfach, Zahlen wiederzugeben. Man muß hier verschiedene Möglichkeiten auseinanderhalten. Zunächst einmal müssen ganze Zahlen (integers) dargestellt werden können, also positive und negative Werte, und schließlich brauchen wir Möglichkeiten zur Darstellung von Dezimalzahlen. Sehen wir uns dazu die Anforderungen und möglichen Lösungen an.

Positive ganze Zahlen lassen sich durch *unmittelbar binäre* Wiedergabe darstellen. Eine solche unmittelbar binäre Darstellung ist im Grunde nichts weiter als die Wiedergabe des dezimalen Werts einer Zahl im Dualsystem. Im Dualsystem gibt die ganz rechts stehende Ziffer den Wert 2^0 wieder, die nächste steht für 2^1, die darauffolgende 2^2 und das ganz links stehende Bit schließlich hat den Wert $2^7 = 128$.

$$b_7b_6b_5b_4b_3b_2b_1b_0$$

steht für

$$b_72^7 + b_62^6 + b_52^5 + b_42^4 + b_32^3 + b_22^2 + b_12^1 + b_02^0$$

Die Zweierpotenzen haben dabei die Werte:
$2^7 = 128$, $2^6 = 64$, $2^5 = 32$, $2^4 = 16$, $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, $2^0 = 1$

Die binäre Wiedergabe entspricht der dezimalen Wiedergabe von Zahlen, wo „123“ für folgendes steht:

$$\begin{array}{rrrr} & 1 \text{ x } 100 & = & 100 \\ + & 2 \text{ x } 10 & = & 20 \\ + & 3 \text{ x } 1 & = & 3 \\ \hline & & = & 123 \end{array}$$

Beachten Sie, daß $100 = 10^2$, $10 = 10^1$ und $1 = 10^0$ ist.

In dieser „Stellenschreibweise“ gibt jede Stelle eine andere Zehnerpotenz wieder. Im Dualsystem steht jedes Bit für eine Zweierpotenz anstatt eine Zehnerpotenz wie im Dezimalsystem.

Ein Beispiel:
Die Dualzahl „00001001“ hat den dezimalen Wert:

$$\begin{array}{lrl} 1 \text{ x } 1 = 1 & & (2^0) \\ 0 \text{ x } 2 = 0 & & (2^1) \\ 0 \text{ x } 4 = 0 & & (2^2) \\ 1 \text{ x } 8 = 8 & & (2^3) \\ 0 \text{ x } 16 = 0 & & (2^4) \\ 0 \text{ x } 32 = 0 & & (2^5) \\ 0 \text{ x } 64 = 0 & & (2^6) \\ 0 \text{ x } 128 = 0 & & (2^7) \\ \hline \quad = 9 & & \end{array}$$

Sehen wir uns noch ein paar Beispiele an:

„10000001“ steht für dezimal:

$$\begin{array}{rrr} 1 \text{ x } & 1 = & 1 \\ 0 \text{ x } & 2 = & 0 \\ 0 \text{ x } & 4 = & 0 \\ 0 \text{ x } & 8 = & 0 \\ 0 \text{ x } & 16 = & 0 \\ 0 \text{ x } & 32 = & 0 \\ 0 \text{ x } & 64 = & 0 \\ 1 \text{ x } & 128 = & 128 \\ \hline & = & 129 \end{array}$$

Also gibt „10000001“ die Dezimalzahl 129 wieder.

Wenn Sie die binäre Wiedergabe von Zahlen untersuchen, wird klar werden, warum man die Bits von rechts nach links mit 0 angefangen durchzählt. Bit 0 ist „b_0“ und entspricht 2^0. Bit 1 ist „b_1“ und gehört zu 2^0 und so weiter.

Dezimal	Binär	Dezimal	Binär
0	00000000	32	00100000
1	00000001	33	00100001
2	00000010	•	
3	00000011	•	
4	00000100	•	
5	00000101	63	00111111
6	00000110	64	01000000
7	00000111	65	01000001
8	00001000	•	
9	00001001	•	
10	00001010	127	01111111
11	00001011	128	10000000
12	00001100	129	10000001
13	00001101		
14	00001110	•	
15	00001111	•	
16	00010000		
17	00010001	•	
•			
•			
•		254	11111110
31	00011111	255	11111111

Bild 1-2: Dezimal/Dual-Umwandlungstabelle

Die binären Entsprechungen der Zahlenwerte von 0 bis 255 zeigt Bild 1-2.

Übung 1.1:
Was ist der dezimale Wert von „11111100“?

Dezimal-Dual-Umwandlung

Lassen Sie uns umgekehrt das binäre Äquivalent von dezimal „11“ errechnen:

$11 : 2 = 5$ Rest $1 \rightarrow 1$ (LSB)
$5 : 2 = 2$ Rest $1 \rightarrow 1$
$2 : 2 = 1$ Rest $0 \rightarrow 0$
$1 : 2 = 0$ Rest $1 \rightarrow 1$ (MSB)

(LSB steht dabei für "least significant bit“ – „niederstwertiges Bit“ und MSB für „most significant bit“ – „höchstwertiges Bit“). Damit lautet die binäre Darstellung von „11“ (gelesen von unten nach oben): 1011.

Man erhält die einer gegebenen Dezimalzahl entsprechende Dualzahl durch schrittweises Teilen der Zahl durch 2, bis der Quotient 0 erreicht ist.

Übung 1.2:
Was ist die binäre Form für 257?

Übung 1.3:
Wandeln Sie 19 in eine Dualzahl um und dann zurück in eine Dezimalzahl.

Rechnen mit Dualzahlen

Die arithmetischen Regeln zum Umgang mit Dualzahlen sind nicht weiter schwierig. Für die Addition lauten sie:

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = (1) 0

Dabei steht (1) für einen „Übertrag" von 1. (Beachten Sie, daß „10" dem dezimalen Wert „2" entspricht!) Binäre Subtraktion wird durch „Addition des Komplements" erledigt und wird erklärt werden, wenn wir erst wissen, wie man negative Zahlen darstellen kann.

Ein Beispiel:

```
  (2)      10
+ (1)    + 01
-------------
= (3)    = 11
```

Diese Addition wird genau wie im dezimalen Fall ausgeführt:
Man addiert spaltenweise, von rechts nach links.

Addition der Spalte ganz rechts:

```
      10
    + 01
    ----
(0 +   1 = 1. Kein Übertrag)
```

Addition der nächsten Spalte:

```
  10
+ 01
----
  11 (1 + 0 = 1. Kein Übertrag)
```

Übung 1.4:
Berechnen Sie 5 + 10 in binärer Form. Prüfen Sie nach, daß das Ergebnis 15 lautet.

Noch ein paar Beispiele zur binären Addition:

```
  0010   (2)        0011   (3)
+ 0001   (1)      + 0001   (1)
-----------       -----------
= 0011   (3)      = 0100   (4)
```

Das letzte Beispiel verdeutlicht die Rolle des Übertrags.

Sehen wir uns die beiden ganz rechts stehenden Bits an:
1 + 1 = (1) 0
Ein Übertrag von 1 ist entstanden, der zu den nächsten Bits hinzuaddiert werden muß:

```
       001 – Spalte 0 ist gerade addiert worden
     + 000 –
     +   1 – (Übertrag)
----------
= (1)  0 – wobei (1) einen weiteren Übertrag in Spalte 2 darstellt.
```

Das Endergebnis lautet: 0100.

Noch ein Beispiel:

```
  0111       (7)
+ 0011   +   (3)
----------------
= 1010   =  (10)
```

Auch in diesem Beispiel wurde ein Übertrag erzeugt und zwar in jeder Spalte bis ganz nach links.

Übung 1.5:
Berechnen Sie das Ergebnis von:

```
  1111
+ 0001
------
= ?
```

Reichen 4 Bits dafür aus?

Man kann mit acht Bits unmittelbar die Zahlen „00000000“ bis „11111111“, d.h. „0“ bis „255“ wiedergeben. Hierbei werden sofort zwei Hindernisse sichtbar. Zum einen können wir so nur positive Zahlen wiedergeben. Zum anderen sind wir beim Einsatz von nur acht Bits auf einen Höchstwert von 255 festgelegt. Wir wollen uns im folgenden diesen Problemen zuwenden.

„Signed Binary“

„Signed Binary“ steht für „vorzeichenbehaftete Dualzahl“. In diesem Fall wird das ganz links stehende Bit zur Anzeige des Vorzeichens der Zahl verwendet. Traditio-

nell wird eine „0“ zur Anzeige einer *positiven,* eine „1“ dagegen für eine *negative* Zahl verwendet. Jetzt steht „11111111“ für −127, während „01111111“ den Wert +127 wiedergibt. Damit können wir positive und negative Zahlen darstellen, haben aber den größten Absolutwert auf 127 eingeschränkt.
Beispiel: „00000001“ steht für +1 (die führende „0“ ist das „+“, gefolgt von „0000001“ = 1). „10000001“ hat den Wert −1 (die führende „1“ steht für „−“).

Übung 1.6:
Wie wird „−5“ in „Signed Binary“ wiedergegeben?

Wenden wir uns dem Problem der *Größenordnung* zu: Um größere Zahlen wiedergeben zu können, muß man mehr Bits verwenden. Wenn wir z. B. sechzehn Bits (zwei Bytes) zur Zahlendarstellung verwenden, so können wir Zahlen im Bereich von −32 K bis +32 K wiedergeben (1 K steht im Computerjargon für 1024). Bit 15 wird für das Vorzeichen benutzt und die verbleibenden 15 Bits geben den absoluten Wert an: 2^{15} = 32 K. Wenn das immer noch nicht ausreicht, so kann man drei oder noch mehr Bytes verwenden. Sollen große ganze Zahlen dargestellt werden, so muß man eine größere Zahl an Bytes zur Wiedergabe einsetzen. Aus diesem Grund bieten die meisten einfachen BASIC-Versionen und andere Sprachen nur einen begrenzten Umfang zur Ganzzahldarstellung an. Auf diese Weise lassen sich die intern gehandhabten Ziffern in einem kürzeren Format wiedergeben. Bessere BASIC-Versionen und bessere Versionen anderer Sprachen bieten die Möglichkeit der Darstellung von mehr Dezimalstellen unter Inkaufnahme von mehr Bytes zur Wiedergabe der Zahlen.
Sehen wir uns ein anderes Problem an, das der Rechengeschwindigkeit. Sagen wir, wir wollten eine Addition im eben eingeführten „Signed Binary“-Format mit den Zahlen „−5“ und „+7“ durchführen.

+ 7 wird wiedergegeben durch:	00000111
− 5 wird wiedergegeben durch:	10000101
Die binäre Summe davon ist:	10001100 oder −12.

Das ist falsch. Das richtige Ergebnis sollte +2 heißen. Um diese Zahldarstellung einsetzen zu können, muß man besondere Operationen durchführen, abhängig vom Vorzeichen der Zahlen. Das führt zu größerer Komplexität der Rechnungen und damit zu verringerter Rechenleistung. Mit anderen Worten: Die rein binäre Addition zweier „Signed Binary“-Zahlen „arbeitet nicht richtig“. Das stört doch ziemlich. Denn es ist ganz klar, der Computer muß die Information nicht nur wiedergeben können, er muß damit auch rechnen.
Die Lösung dieses Problems wird als *Zweierkomplementdarstellung* bezeichnet, die anstelle der „Signed Binary“-Form benutzt wird. Um dieses Zweierkomplement einzuführen, wollen wir erst einen Zwischenschritt machen und uns das sogenannte *Einerkomplement* ansehen.

Das Einerkomplement

Im Einerkomplementformat werden die positiven ganzen Zahlen alle im gewohnten Binärformat dargestellt. „+3“ z. B. wird wie üblich als 00000011 wiedergegeben. Sein

Komplement jedoch, „−3“, erhält man, indem man jedes Bit der Originaldarstellung komplementiert, Jede 0 wird eine 1 und jede 1 in eine 0 übergeführt. In unserem Beispiel würde „−3“ durch die Einerkomplementzahl 11111100 dargestellt.

Noch ein Beispiel:

+2 lautet 00000010
−2 lautet 11111101

Beachten Sie, daß in dieser Darstellung positive Zahlen links mit einer „0“, negative mit einer „1“ eingeleitet werden.

Übung 1.7:
„+6“ wird als „00000110“ wiedergegeben. Wie sieht die Darstellung von „−6“ im Einerkomplement aus?

Addieren wir zur Probe die Zahlen „−4“ und „+6“:

− 4 lautet 11111011
+ 6 lautet 00000110

Das ergibt: (1) 00000001, wobei (1) einen Übertrag bezeichnet.

Das richtige Ergebnis sollte dagegen „+2“ oder „00000010“ sein.

Probieren wir das noch einmal:

−3 lautet 11111100
−2 lautet 11111101

Das ergibt: (1) 00000001

oder „1“ und einen Übertrag. Das richtige Ergebnis wäre aber „−5“. Dieser Wert wird jedoch durch 11111010 wiedergegeben. Es hat wieder nicht funktioniert.
Dieses Zahlenformat vermag positive und negative Zahlen darzustellen. Eine normale Addition bringt jedoch nicht immer das „richtige“ Ergebnis. Wir müssen noch eine andere Darstellung finden. Sie ergibt sich aus dem Einerkomplement und wird Zweierkomplement genannt.

Zweierkomplementdarstellung

In der Zweierkomplementdarstellung werden positive Zahlen wie beim Einerkomplement der „Signed Binary“-Form entsprechend wiedergegeben. Der Unterschied liegt in der Wiedergabe von *negativen Zahlen.* Eine negative Zahl in Zweierkomplementdarstellung ergibt sich, indem man erst das Einerkomplement ermittelt und dann *eine Eins addiert.* Untersuchen wir dazu ein Beispiel:
+3 wird im „Signed Binary“-Format als 00000011 wiedergegeben. Sein Einerkomplement lautet 11111100. Die Zweierkomplementform erhält man durch Addition von Eins. Sie lautet 11111101.

Versuchen wir damit eine Addition:

(3)	00000011
+ (5)	+ 00000101
= (8)	= 00001000

Das Ergebnis ist in Ordnung. Sehen wir uns also eine Subtraktion an:

(3)	00000011
− (5)	+ 11111011
	= 11111110

Den Wert des Ergebnisses können wir durch Berechnen seines Zweierkomplements herausbekommen:

Einerkomplement von 11111110:	00000001
1 dazu addieren:	+ 00000001

Das ergibt als Zweierkomplement: 00000010 oder +2.

Unser Ergebnis von eben, 11111110, stellt damit „−2" dar. Es ist korrekt.
Wir haben damit die Addition und die Subtraktion von Zweierkomplementzahlen ausprobiert und die richtigen Ergebnisse erhalten. Es sieht so aus, als würde Zweierkomplementrechnung funktionieren!

Übung 1.8:
Wie lautet die Zweierkomplementdarstellung von „+127"?

Übung 1.9:
Wie lautet die Zweierkomplementdarstellung von „−128"?

Addieren wir jetzt +4 und −3 (die Subtraktion erfolgt immer durch Addieren des Zweierkomplements):

	+4 lautet 00000100
	−3 lautet 11111101
Das ergibt:	(1) 00000001

wobei (1) einen Übertrag andeutet. Wenn wir den Übertrag ignorieren, erhalten wir 00000001, also den dezimalen Wert „1". Damit ist das Ergebnis richtig. Ohne einen vollständigen mathematischen Beweis zu geben, lassen sie uns einfach nur festhalten, daß die Zweierkomplementrechnung funktioniert. Benutzt man die Zweierkomplementdarstellung, so kann man vorzeichenbehaftete Zahlen ohne Untersuchung des Vorzeichens einfach addieren und subtrahieren. Allein durch Anwendung der üblichen Additionsregeln für Dualzahlen erhält man das richtige Ergebnis, *einschließlich des Vorzeichens.* (Der Übertrag wird dabei ignoriert.) Das ist ein sehr wesentlicher

Vorzug. Wenn dies nicht möglich wäre, so müßte man jedesmal das Ergebnis je nach Vorzeichen korrigieren, was die Additions- bzw. Subtraktionszeit stark verlängern würde.
Der Vollständigkeit halber sei angemerkt, daß die Zweierkomplementform einfach nur die bequemste Wiedergabemöglichkeit für den Einsatz einfacherer Rechner wie der Mikroprozessoren ist. Komplexere Maschinen können andere Darstellungen einsetzen. So wird z. B. das Einerkomplement angewendet, wobei der Prozessor über besondere Schaltungen zur „Ergebniskorrektur" verfügt.
Von nun an wollen wir alle vorzeichenbehaftete Zahlen ohne weitere Angaben als in Zweierkomplementform dargestellt verstehen. Bild 1-3 zeigt eine Tabelle von Zweierkomplementzahlen.

Übung 1.10:
Wie lautet die größte und wie die kleinste im Zweierkomplement mit nur einem Byte darstellbare Zahl?

Übung 1.11:
Berechnen Sie das Zweierkomplement von 20. Ermitteln Sie darauf das Zweierkomplement Ihres Ergebnisses. Erhalten Sie 20 zurück?

Mit den folgenden Beispielen sollen die Regeln der Zweierkomplementrechnung verdeutlicht werden. Insbesondere stellt C einen etwa eingetretenen Übertragszustand („carry") dar. (Es handelt sich dabei um Bit 8 des Ergebnisses.)
V bezeichnet einen Zweierkomplementüberlauf („overflow"), d. h. die Tatsache, daß das Vorzeichen des Ergebnisses „versehentlich" umgekehrt worden ist, weil die in die Rechnung eingehenden Zahlen zu groß waren. Im Grunde handelt es sich dabei um einen internen Übertrag von Bit 6 in Bit 7 (das Vorzeichenbit). Das wird unten genauer dargestellt werden.
Sehen wir uns die Rolle des Übertrag „C" und des Überlaufs „V" näher an.

Der Übertrag (carry) C

Hier ist ein Beispiel für einen Übertrag:

```
    (128)          10000000
+   (129)     +    10000001
---------------------------
    (257)  =   (1) 00000001
```

wobei (1) für den Übertrag steht.

Das Ergebnis erfordert ein neuntes Bit (Bit „8", da das ganze rechts stehende Bit die Nummer „0" trägt). Es ist das Übertragsbit.
Wenn wir den Übertrag als das neunte Bit des Ergebnisses auffassen, dann ergibt sich das Ergebnis zu 100000001 = 257.
Der Übertrag muß allerdings erkannt und mit Vorsicht gehandhabt werden. Innerhalb des Mikroprozessors sind die zur Informationsdarstellung erforderlichen Regi-

ster in der Regel nur acht Bits breit. Wenn man dort das Ergebnis speichert, bleiben nur Bit 0 bis 7 erhalten.
Ein Übertrag erfordert daher immer eine besondere Operation: Er muß durch besondere Befehle erkannt und daraufhin verarbeitet werden. Diese Übertragsverarbeitung bedeutet entweder sein Speichern irgendwo im Prozessor (mit Hilfe eines besonderen Befehles) oder sein Unterschlagen oder die Entscheidung, daß es sich um einen Fehler handelt (wenn das größte erlaubte Ergebnis „11111111" beträgt).

Der Überlauf (overflow) V

Hier ist ein Beispiel für einen Überlauf:

```
Bit 6________↓
Bit 7______↓
        01000000          (64)
    +   01000001        + (65)
    ------------------------------
    =   10000001   =   (−127)
```

+	2er-Komplement Kode	−	2er-Komplement Kode
+127	01111111	−128	10000000
+126	01111110	−127	10000001
+125	01111101	−126	10000010
...		−125	10000011
		...	
+65	01000001	−65	10111111
+64	01000000	−64	11000000
+63	00111111	−63	11000001
...		...	
+33	00100001	−33	11011111
+32	00100000	−32	11100000
+31	00011111	−31	11100001
...		...	
+17	00010001	−17	11101111
+16	00010000	−16	11110000
+15	00001111	−15	11110001
+14	00001110	−14	11110010
+13	00001101	−13	11110011
+12	00001100	−12	11110100
+11	00001011	−11	11110101
+10	00001010	−10	11110110
+9	00001001	−9	11110111
+8	00001000	−8	11111000
+7	00000111	−7	11111001
+6	00000110	−6	11111010
+5	00000101	−5	11111011
+4	00000100	−4	11111100
+3	00000011	−3	11111101
+2	00000010	−2	11111110
+1	00000001	−1	11111111
+0	00000000		

Bild 1-3: 8-Bit-Zweierkomplementzahlen

Es wurde ein interner Übertrag von Bit 6 nach Bit 7 erzeugt. So etwas wird als Überlauf bezeichnet.
Das Ergebnis ist jetzt „versehentlich“ negativ geworden. So ein Fall muß erkannt werden können, damit er zu korrigieren ist.

Sehen wir uns einen anderen Fall an:

```
              11111111         (-1)
            + 11111111     +   (-1)
            ----------------------
= (1)         11111110     =   (-2)
   ↓
Übertrag
```

In diesem Fall ist ein interner Übertrag von Bit 6 nach Bit 7 und ein äußerer Übertrag (der formale „carry“ C aus dem vorigen Schritt) von Bit 7 nach Bit 8 aufgetreten. Die Regeln der Zweierkomplementarithmetik besagen, daß ein solcher Übertrag C unterschlagen werden soll. In diesem Fall ist das Ergebnis in Ordnung.
Das liegt daran, daß der Übertrag von Bit 6 nach Bit 7 das Vorzeichenbit nicht verändert hat.
Es handelt sich hier demzufolge nicht um einen *Überlaufzustand*. Wenn man mit negativen Zahlen arbeitet, ist der Überlauf nicht einfach nur ein Übertrag von Bit 6 nach Bit 7. Sehen wir uns dazu ein weiteres Beispiel an.

```
              11000000        (- 64)
            + 10111111     +  (- 65)
            ----------------------
= (1)         01111111     =  (+127)
   ↓
Übertrag
```

Diesmal gab es keinen internen Übertrag von Bit 6 nach Bit 7, dafür aber einen äußeren von Bit 7 nach Bit 8. Das Ergebnis ist falsch, da das Vorzeichen umgekehrt worden ist. Also muß ein Überlaufzustand angezeigt werden.

Ein Überlauf tritt in folgenden vier Situationen auf:

1 – Addieren von großen positiven Zahlen
2 – Addieren von (absolut) großen negativen Zahlen
3 – Subtrahieren einer großen positiven von einer (absolut) großen negativen Zahl
4 – Subtrahieren einer (absolut) großen negativen Zahl von einer großen positiven Zahl.

Verbessern wir damit unsere Überlaufdefinition:

Rein technisch wird der Überlaufanzeiger, ein speziell für diesen Zweck reserviertes und „Flagge“ (flag) genanntes Bit, gesetzt, wenn ein Übertrag von Bit 6 nach Bit 7

und kein externer Übertrag oder wenn ein externer Übertrag, aber kein interner von Bit 6 nach Bit 7, eingetreten ist. Dadurch wird angezeigt, daß Bit 7, das Vorzeichen des Ergebnisses, versehentlich geändert worden ist. Für den technish interessierten Leser sei gesagt, daß das Überlaufbit durch EXKLUSIV-ODER-Verknüpfung des in Bit 7 eingehenden Übertrags ermittelt wird. Praktisch jeder Mikroprozessor ist mit einer besonderen Überlaufsflagge versehen, die automatisch diesen Zustand erkennt, der spezielle Korrekturoperationen verlangt.
Überlauf gibt an, daß das Ergebnis einer Addition oder einer Subtraktion mehr Bits benötigt, als im Standardregister von acht Bit Breite zur Aufnahme des Ergebnisses bereitstehen.

Übertrag und Überlauf

Übertrags- und Überlaufsbit werden „Flaggen“ genannt. Jeder Mikroprozessor stellt solche Flaggen zur Verfügung, und im übernächsten Kapitel werden wir erfahren, wie man sie zur effektiven Programmierung einsetzen kann. Diese beiden Anzeiger sind in einem speziellen Register untergebracht, das Flaggen- oder „Statusregister“ genannt wird. Dieses Register enthält noch mehr Flaggen, deren Funktionen beim 6502 in Kapitel 4 dargestellt wird.

Beispiele:

Betrachten wir nun die Funktionsweise von Übertrag C und Überlauf V in einigen Beispielen. Wenn dort kein Überlauf aufgetreten ist, haben wir V = 0. Ist ein Überlauf entstanden, so ist V = 1. Dasselbe gilt für das Übertragsbit C. Beachten Sie, daß nach den Regeln der Zweierkomplementsrechnung der Übertrag ignoriert werden muß. (Einen mathematischen Beweis wollen wir hier jedoch dafür nicht liefern.)

Positiv-Positiv

```
  00000110      (+  6)
+ 00001000      (+  8)
-------------------------
= 00001110      (+ 14) V : 0     C : 0
```

(RICHTIG)

Positiv-Positiv mit Überlauf

```
  01111111      (+ 127)
+ 00000001      (+   1)
-------------------------
= 10000000      (- 128) V : 1     C : 0
```

(FALSCH)

Positiv-Negativ (Ergebnis positiv)

```
  00000100        (+ 4)
+ 11111110        (- 2)
-----------------------
= (1) 00000010    (+ 2) V : 0    C : 1 (ignorieren)
```

(RICHTIG)

Positiv-Negativ (Ergebnis negativ)

```
  00000010        (+ 2)
+ 11111100        (- 4)
-----------------------
= 11111110        (- 2) V : 0    C : 0
```

(RICHTIG)

Negativ-Negativ

```
  11111110        (- 2)
+ 11111010        (- 4)
-----------------------
= (1) 11111000    (- 6) V : 0    C : 1 (ignorieren)
```

(RICHTIG)

Negativ-Negativ mit Überlauf

```
  10000001        (- 127)
+ 11000010        (-  62)
-------------------------
= (1) 01000011       (67) V : 1    C : 1
```

(FALSCH)

Diesmal ist durch Addieren zweier absolut großer negativer Zahlen ein „Unterlauf" eingetreten. Das Ergebnis wäre −189, was dem Absolutbetrag nach zu groß für acht Bits ist.

Übung 1.12:
Vervollständigen Sie die folgenden Additionen. Geben Sie das Ergebnis, den Übertrag C, den Überlauf V sowie die Tatsache an, ob das Ergebnis richtig oder falsch ist:

```
  10111111        (__)                 11111010        (__)
+ 11000001        (__)               + 11111001        (__)
-----------                          -----------

=           V: ____   C: ____        =           V: ____   C: ___
-----------                          -----------
( ) RICHTIG     ( ) FALSCH           ( ) RICHTIG   ( ) FALSCH
```

```
  00010000      (___)
+ 01000000      (___)

=           V: ___   C: ___
____________
( ) RICHTIG     ( ) FALSCH
```

```
  01111110      (___)
+ 00101010      (___)

=           V: ___   C: ___
____________
( ) RICHTIG     ( ) FALSCH
```

Übung 1.13:
Können Sie ein Beispiel eines Überlaufs beim Addieren einer positiven und einer negativen Zahl geben? Warum ist das so?

Festformatdarstellung

Wir wissen jetzt, wie man vorzeichenbehaftete ganze Zahlen wiedergeben kann. Jedoch haben wir das Problem der Größenordnung noch immer nicht gelöst. Wenn wir größere Zahlen darstellen wollen, benötigen wir mehrere Bytes. Um arithmetische Operationen effektiv ausführen zu können, ist es demgegenüber notwendig, statt einer variablen eine feste Anzahl von Bytes zur Zahlendarstellung heranzuziehen. Wenn daher die Byteanzahl erst einmal festgelegt ist, liegt auch die Größenordnung der darstellbaren Zahlen fest.

Übung 1.14:
Was ist der Wert der größten und der kleinsten mit zwei Bytes im Zweierkomplement darstellbaren Zahl?

Problem der Größenordnung
Wir haben uns bei der Addition von Zahlen auf acht Bites beschränkt, da der Prozessor intern nur acht Bites auf einmal verarbeiten kann. Das engt den verfügbaren Zahlenbereich jedoch auf −128 bis +127 ein, was für viele Anwendungen nicht ausreicht. Um die Zahl der darstellbaren Stellen zu erhöhen, ist eine sogenannte mehrfache Genauigkeit notwendig. Man kann dazu ein Zwei-, Drei- oder Mehrbyteformat einsetzen. Betrachten wir z.B. das 16 Bit umfassende Format mit „doppelter Genauigkeit":

00000000 00000000	ist „0"
00000000 00000001	ist „1"
01111111 11111111	ist „32767"
11111111 11111111	ist „−1"
11111111 11111110	ist „−2"

Übung 1.15:
Was ist die dem Betrag nach größte negative ganze Zahl, die mit dreifacher Genauigkeit dargestellt werden kann?

Diese Methode hat jedoch einige Nachteile. Wenn man z.B. zwei Zahlen addieren möchte, muß man das Byte für Byte tun. Das wird in Kapitel 3 (Grundlegende Techniken der Programmierung) genauer beschrieben werden. Das wiederum bewirkt ei-

ne langsamere Ausführung der Rechnung. Des weiteren benutzt diese Darstellung für jede Zahl 16 Bits, selbst wenn acht Bits voll ausreichen würden. Es ist daher üblich, Zahlen von 16 Bits Breite, vielleicht auch von 32 Bits, selten aber längere zu benutzen.
Dabei ist folgender wichtiger Umstand zu berücksichtigen: Was auch immer die Anzahl N ist, die wir für die zur Zahldarstellung verwendeten Bits gewählt haben – einmal ausgewählt liegt sie unverrückbar fest. Wenn irgendein Zwischenergebnis einer Rechnung mehr als diese N Bits benötigen sollte, werden einige Bits verlorengehen. Üblicherweise sichert ein Programm in so einem Fall die (höchstwertigen) linken Bits und verliert die überschüssigen rechts stehenden. Man nennt dies das „Stutzen" (truncating) des Ergebnisses.

Hier ist ein Beispiel im Dezimalsystem, bei dem sechs Stellen zur Darstellung verwendet werden sollen:

```
123456 x 1,2
------------
    123456
+    246912
------------
= 148147,2
```

Das Ergebnis fordert 7 Stellen! Das heißt, daß die „2" nach dem Komma wieder verlorengeht und wir als Ergebnis 148147 erhalten. Es ist gestutzt worden. Man benutzt diese Methode üblicherweise, solange das Komma nicht verlorengeht, um den Bereich der ausführenden Operationen auf Kosten der Genauigkeit zu erweitern.
Bei Dualzahen haben wir dasselbe Problem. Die Einzelheiten zur binären Multiplikation werden in Kapitel 3 gezeigt.
Eine solche Festformatdarstellung mag einen Genauigkeitsverlust bewirken, aber sie kann für die üblichen Rechenanforderungen und für mathematische Zwecke ausreichen.
Leider gibt es – beispielsweise in der Buchhaltung – Fälle, in denen ein derartiger Genauigkeitsverlust nicht zugelassen werden kann. Wenn z.B. für einen Kunden auf der Registrierkasse ein großer Betrag zusammenkommt, dann ist es nicht sinnvoll, ihn einen nur fünfstelligen, auf die Mark abgerundeten Betrag zahlen zu lassen. Wo immer die Genauigkeit des Ergebnisses wichtig ist, muß eine andere Zahldarstellung verwendet werden. Üblicherweise löst man dasProblem mit BCD-Zahlen.

Die BCD-Darstellung

BCD steht für „binary-coded decimal" – „binär kodierte Dezimalziffern". Das zur Wiedergabe von Zahlen im BCD-Kode benutzte Prinzip sagt, daß jede Stelle für sich zu kodieren ist und daß so viele Bits wie zur vollständigen Zahldarstellung notwendig benutzt werden. Um alle Ziffern von 0 bis 9 zu erfassen, benötigt man vier Bits. Drei Bits bieten nur acht Möglichkeiten und können daher die zehn Ziffern nicht verschlüsseln. Mit vier Bits hat man sechzehn Möglichkeiten und damit genug zum Kodieren der Ziffern „0" bis „9". Dabei ist festzuhalten, daß sechs von den mit vier Bits möglichen Kombinationen bei der BCD-Darstellung nicht benutzt werden (siehe

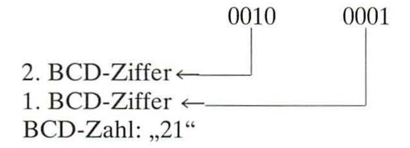
Bild 1-4). Das ergibt später Probleme bei der Addition und Subtraktion, die wir in den Griff bekommen müssen. Da zur Verschlüsselung einer BCD-Stelle nur vier Bits benötigt werden, kann man zwei Stellen in einem Byte unterbringen, was man mit *gepackter BCD-Darstellung* bezeichnet.

KODE	BCD SYMBOL	KODE	BCD SYMBOL
0000	0	1000	8
0001	1	1001	9
0010	2	1010	*unbenutzt*
0011	3	1011	*unbenutzt*
0100	4	1100	*unbenutzt*
0101	5	1101	*unbenutzt*
0110	6	1110	*unbenutzt*
0111	7	1111	*unbenutzt*

Bild 1-4: BCD-Kodes

Zum Beispiel hat „00000000“ als BCD-Zahl den Wert „00“ und „10011001“ steht für „99“.

Ein BCD-kodiertes Byte liest man wie folgt:

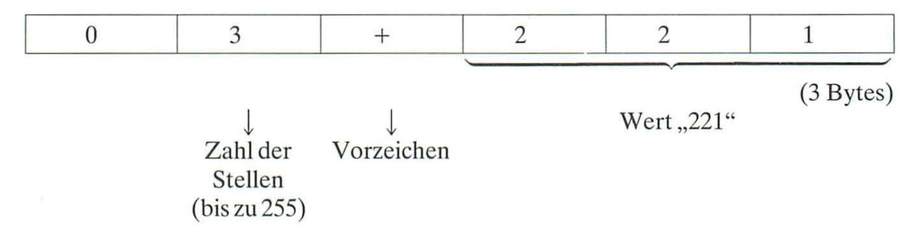


Übung 1.16:
Wie lautet die BCD-Darstellung von „29“? Wie die von „91“?

Übung 1.17:
Ist „10100000“ eine gültige BCD-Darstellung? Warum?

Zur Wiedergabe einer Zahl in BCD-Form werden so viele Bytes wie zur Darstellung aller Stellen nötig benutzt.
Üblicherweise setzt man ein oder zwei Nibbles am Anfang der Darstellung ein, mit denen die Anzahl der insgesamt verwendeten BCD-Ziffern angegeben wird. Ein weiteres Nibble kann angeben, an welcher Stelle sich das Komma befindet. Im wesentlichen ist das jedoch Vereinbarungssache.

Hier ist ein Befehl für die Darstellung einer mehrere Bytes umfassenden ganzen Zahl in BCD-Form:



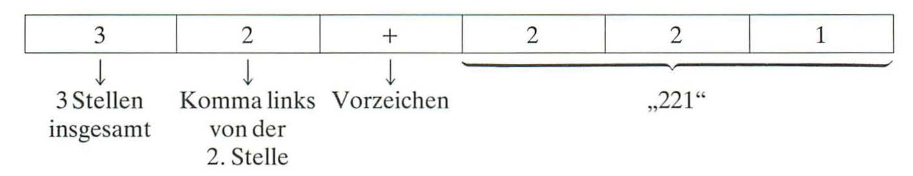
Damit wird die Zahl „+221“ wiedergegeben.
(Das Vorzeichen kann z.B. durch 0000 für „+“ und durch 0001 für „−“ dargestellt werden.

Übung 1.18:
Stellen Sie nach derselben Vereinbarung die Zahl „−23123“ dar. Geben Sie sie sowohl in BCD-Form, als auch als Zweierkomplementzahl wieder.

Übung 1.19:
Stellen Sie die BCD-Zahlen „222“ und „111“ dar und dann das Ergebnis der Multiplikation 222 x 111. (Berechnen Sie das Produkt von Hand und wandeln Sie es anschließend in die oben angegebene Darstellung um).

Mit der BCD-Darstellung lassen sich Dezimalzahlen ganz einfach erfassen.

Die Zahl „+2,21“ kann man z.B. so wiedergeben:



Der Vorteil von BCD-Zahlen liegt darin, daß sie vollkommen richtige Ergebnisse liefern können. Der Nachteil ist die Tatsache, daß zu ihrer Darstellung sehr viel Speicherplatz benötigt wird und daß die Rechenoperationen relativ langsam ablaufen. Das läßt sich nur im kaufmännischen Bereich tolerieren, in anderen Fällen verwendet man dieses Format nur selten.

Übung 1.20:
Wie viele Bits braucht man zur Darstellung der BCD-Zahl „9999“ und wieviele im Zweierkomplementformat?

Wir haben damit die Probleme im Zusammenhang mit der Darstellung von ganzen Zahlen mit und ohne Vorzeichen und sogar von großen ganzen Zahlen gelöst. Außer-

dem kennen wir mittlerweile mit der BCD-Darstellung eine Möglichkeit, Dezimalzahlen wiederzugeben. Betrachten wir nun das Problem, Dezimalzahlen mit einer festen Anzahl von Bytes wiedergeben zu müssen.

Gleitkommandodarstellung

Das Grundprinzip ist hier, daß die Dezimalzahlen mit einem festen Format wiedergegeben werden müssen. Um hierbei keine Bits zu verschenken, werden in der Darstellung alle Zahlen *normalisiert.*
So verschenkt z.B. „0,000123" hinter dem Komma drei Nullen, die keine andere Bedeutung haben, als die Stelle des Kommas anzugeben. Normalisieren dieser Zahl ergibt die Darstellung „$0{,}123 \times 10^{-3}$". „0,123" wird als *normalisierte Mantisse,* „−3" als *Exponent* bezeichnet. Wir haben diese Zahl durch Streichen der bedeutungslosen Nullen links von ihr und anschließendes Anpassen des Exponenten gewonnen.

Betrachten wir ein anderes Beispiel:

„22,1" wird zu „$0{,}221 \times 10^{2}$" normalisiert.

Allgemein erhält eine normalisierte Zahl die Form $M \times 10^{E}$, wobei M die Mantisse und E der Exponent ist.
Man sieht daraus, daß eine normalisierte Zahl durch eine Mantisse zwischen 1 und 0,1 gekennzeichnet ist, sofern es sich nicht um den Wert Null handelt. Mit anderen Worten läßt sich dies mathematisch so darstellen:

$$0{,}1 \leq M < 1 \text{ oder } 10^{-1} \leq M < 10^{0}$$

Entsprechendes gilt im binären Fall:

$$2^{-1} \leq M < 2^{0} \text{ (oder: } 0{,}5 \leq M < 1\text{)}$$

Dabei ist M der Absolutbetrag der Mantisse (d.h. ihr Wert ohne Vorzeichen).

Ein Beispiel:

$$111{,}01 \text{ wird normalisiert zu } 0{,}111011 \times 2^{3}$$

In der Darstellung kann das „0," auch noch wegfallen, so daß die Mantisse die Form 11101 und der Exponent den Wert 3 bekommt (siehe Bild 1-5).
Nachdem wir damit das Prinzip der Darstellung definiert haben, wollen wir uns ein Format aus der Praxis ansehen. Eine typische Gleitkommadarstellung findet sich in Bild 1-5.

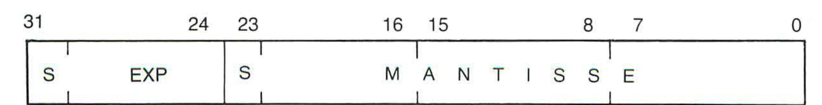


Bild 1-5: Ein typisches Gleitkommaformat

Bei der in diesem Beispiel benutzten Darstellung werden vier Bytes für eine Gesamtlänge der Zahl von 32 Bits eingesetzt. Das erste Byte links in der Abbildung dient zur Wiedergabe des Exponenten. Sowohl Exponent als auch Mantisse sind im Zweierkomplement dargestellt. Damit kann man als kleinsten Exponenten den Wert „–128", als größten den Wert „+127" wiedergeben. Das „S" in Bild 1-5 bezeichnet das Vorzeichenbit (sign bit).
Zur Wiedergabe der Mantisse werden drei Bytes benutzt. Da das erste Bit einer Zweierkomplementzahl für das Vorzeichen dient, bleiben zur Wiedergabe des Betrags der Zahl gerade 23 Bits übrig.

Übung 1.21:
Wie viele Dezimalstellen lassen sich mit einer Mantisse von 23 Bits darstellen?

Das ist nur ein Beispiel für Gleitkommaformat. Man kann auch nur drei Bytes benutzen, genauso wie man bei Bedarf noch mehr Bytes einsetzen kann. Die oben vorgeschlagene Vierbytedarstellung ist jedoch am weitesten verbreitet, da sie einen sinnvollen Kompromiß zwischen Genauigkeit, Größenordnung der Zahl, Speicherbedarf und Effektivität der Rechenoperation darstellt.
Wir haben damit die Probleme untersucht, die bei der Wiedergabe von Zahlen im Rechner auftreten können, und wir wissen, wie man ganze Zahlen mit und ohne Vorzeichen oder Dezimalzahlen darstellt. Sehen wir uns im folgenden an, wie man alphanumerische Daten im Computer wiedergeben kann.

Darstellung alphanumerischer Daten

Die Darstellung alphanumerischer Daten, d. h. von Buchstaben, Ziffern und Zeichen, stellt keine besonderen Probleme: Alle Zeichen werden in einem Achtbitkode verschlüsselt. Dabei sind im Computerbereich im wesentlichen nur zwei Kodes weithin gebräuchlich, der ASCII-Kode und der EBCDIC-Kode. ASCII steht für „American Standard Code for Information Interchange" (Amerikanischer Standardkode zum Informationsaustausch; international als ISO-7-Bit-Kode genormt). Im Mikroprozessorbereich wird dieser Kode fast ausschließlich eingesetzt. EBCDIC ist eine von IBM entwickelte und eingesetzte Verschlüsselung und kaum mehr außerhalb dieses Bereichs gebräuchlich. Im Mikrocomputerbereich begegnet man diesem Kode höchstens beim Versuch, ein IBM-Peripheriegerät anzuschließen.
Sehen wir uns die ASCII-Verschlüsselung kurz an. Wir müssen hierbei 26 Buchstaben in Groß- und Kleinschreibung erfassen, dazu 10 Ziffern und vielleicht 20 Sonderzeichen. Das läßt sich bequem mit 7 Bits erreichen, die 128 Kombinationsmöglichkeiten bieten (vgl. Bild 1-6). Alle Zeichen werden daher mit sieben Bits verschlüsselt. Das achte Bit dient, wenn überhaupt benutzt, als *Paritätsbit* (parity bit). Parität ist ein Verfahren, mit dem nachgeprüft werden kann, daß der Inhalt des Bytes nicht versehentlich geändert worden ist. Man zählt die Einsen im Byte und setzt das achte Bit auf Eins, wenn eine ungerade Anzahl vorgefunden wurde. Auf diese Weise erhält man insgesamt eine gerade Anzahl von Einsen im Byte. So etwas nennt man gerade Parität (even parity). Man kann genausogut auch ungerade Parität benutzen, also das achte Bit (d.h. das ganz links stehende) so setzen, daß insgesamt eine ungerade Anzahl von Einsen im Byte steht.

Ein Beispiel: Ermitteln wir das Paritätsbit für „0010011“ mit gerader Parität. Wir haben 3 Einsen vorliegen. Das Paritätsbit muß damit auf eine 1 gesetzt werden, um auf eine gerade Anzahl von 4 Einsen zu kommen. Es ergibt sich so 10010011 mit einer führenden 1 als Paritätsbit und 0010011 als Wiedergabe des Zeichens selbst.
Bild 1-6 zeigt die Bedeutung der verschiedenen Siebenbitmuster im ASCII-Kode. In der Praxis verwendet man diese Werte „wie sie sind“, d.h. ohne Paritätsbit, indem man einfach ganz links eine 0 anfügt. Seltener findet man die ausdrückliche Verwendung der Paritätsmöglichkeit, wobei dann auf der linken Seite das passende Paritätsbit angefügt wird.

Übung 1.22:
Berechnen Sie die 8-Bit-Formen der Ziffern „0“ bis „9“ mit gerader Parität. (Diese Darstellung brauchen wir in einigen Anwendungsbeispielen in Kapitel 8).

Übung 1.23:
Machen Sie dasselbe mit den Buchstaben „A“ bis „E“.

BIT NUMBERS															
								0	0	0	0	1	1	1	1
								0	0	1	1	0	0	1	1
								0	1	0	1	0	1	0	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	HEX 1 / HEX 0	0	1	2	3	4	5	6	7
			0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
			0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
			0	0	1	0	2	STX	DC2	"	2	B	R	b	r
			0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
			0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
			0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
			0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
			0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
			1	0	0	0	8	BS	CAN	(	8	H	X	h	x
			1	0	0	1	9	HT	EM	)	9	I	Y	i	y
			1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
			1	0	1	1	11	VT	ESC	+	;	K	[	k	{
			1	1	0	0	12	FF	FS	,	<	L	\	l	¦
			1	1	0	1	13	CR	GS	-	=	M	]	m	}
			1	1	1	0	14	SO	RS	.	>	N	^	n	~
			1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Bild 1-6: Der ASCII-Kode

Übung 1.24:
Entschlüsseln Sie die folgenden 5 Bytes unter der Annahme, daß es sich bei ihnen um einen paritätslosen ASCII-Kode (bei dem das ganz links stehende Bit eine „0" ist) handelt:

01001000 01100001 01101100 01101111 00100001

In besonderen Situationen, wie z.B. bei der Datenfernverarbeitung können auch andere Verschlüsselungen, wie z.B. fehlerkorrigierende Kodes verwendet werden. Diese liegen jedoch außerhalb des Rahmens unseres Buches.
Wir haben damit sowohl die interne Darstellung von Programmen als auch die von Daten im Computer untersucht. Sehen wir uns noch die Möglichkeiten zur externen Wiedergabe an.

Externe Informationsdarstellung

Externe Darstellung bezieht sich auf die Art und Weise, in der die Information dem *Benutzer,* in erster Linie dem Programmierer, übermittelt wird. Die Information kann im wesentlichen in drei Formaten ausgegeben werden: binär, oktal oder hexadezimal und symbolisch.

1. Binär

Wir haben gesehen, daß die Information intern in Form von Bytes, also das Folgen von acht *Bits* (Nullen oder Einsen) gespeichert wird. Manchmal ist es erwünscht, diese interne Information unmittelbar im Binärformat dargestellt zu sehen, was man als *Binärdarstellung* bezeichnet. Ein einfaches Beispiel ist die Verwendung von Leuchtdioden (LEDs), bei denen es sich im Prinzip um kleine „Lämpchen" handelt, und die auf der Frontplatte („Konsole") des Mikrocomputers angebracht sind. Im Fall eines Achtbiteprozessors enthält die Konsole üblicherweise acht derartige LEDs zur Darstellung des Inhalts beliebiger interner Register. (Ein Register wird zum Festhalten von acht Informationsbits benutzt und in Kapitel 2 näher beschrieben). Eine leuchtende LED zeigt eine Eins an. Eine Null wird durch eine nichtleuchtende LED wiedergegeben. Man kann eine derartige Binärdarstellung zum Aufspüren von Detailfehlern in einem komplexen Programm benutzen, insbesondere, wenn Ein- und Ausgabebausteine mit einbezogen sind. Allerdings ist diese Technik dem Menschen nicht sehr angepaßt. Das liegt daran, daß in den meisten Fällen nicht das Bitmuster, sondern die mit diesem dargestellte Information untersucht werden soll, die man besser in symbolischer Form erkennt. Eine „9" ist viel leichter zu erkennen als das Muster „1001". So sind leichter verständliche Darstellungsformen entwickelt worden, die die Zusammenarbeit Mensch-Maschine verbessern helfen.

2. Oktal und hexadezimal

„Oktal" und „hexadezimal" fassen drei bzw. vier Bits zu einem Zeichen zusammen. Im Oktalsystem wird jede Kombination von drei Bits als Ziffer zwischen 0 und 7 dargestellt (siehe Bild 1-7).

binär	oktal
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Bild 1-7: Die Oktalsymbole

So wird z.B. das binäre „00 100 100“ wiedergegeben als:

⇩ ⇩ ⇩
0 4 4

d.h. oktal „044“.

Ein anderes Beispiel:

11 111 111

ergibt

⇩ ⇩ ⇩
3 7 7

d.h. oktal „377“.

Umgekehrt entspricht dem oktalen „211“ die binäre Form

2 1 1
⇩ ⇩ ⇩
10 001 001

also „10001001“.

Oktal wurde traditionell in älteren Computern benutzt, die Wortlängen im Bereich von 8 bis etwa 64 verwendeten. Mit dem Durchbruch von Achtbitmikroprozessoren hat sich jedoch in letzter Zeit das Achtbitformat als Standard durchgesetzt, für das mit dem *hexadezimalen* Format eine besser angepaßte Darstellung verwendet wird.

In der Hexadezimaldarstellung wird jeweils eine Gruppe von vier Bits zu einer Hexadezimalziffer zusammengefaßt. Diese Hexadezimalziffern werden durch die Zeichen „0“ bis „9“ und „A“ bis „F“ wiedergegeben. So steht z.B. „0“ für „0000“, „1“ für „0001“ und „F“ für „1111“ (siehe Bild 1-8).

Ein Beispiel:
Der binäre Wert 1010 0001 wird hexadezimal als

⇩ ⇩
A 1

„A1“ wiedergegeben.

Übung 1.25:
Wie lautet die hexadezimale Form von „10101010"?

Übung 1.26:
Was für einer binären Darstellung entspricht umgekehrt das hexadezimale „FA"?

Übung 1.27:
Wie wird „01000001" oktal wiedergegeben?

Hexadezimalzahlen bieten den Vorzug, acht Bits in zwei Ziffern zusammenfassen zu können. Das läßt sich rascher erkennen, merken und in den Computer eingeben als das binäre Äquivalent. Aus diesem Grund wird die Hexadezimaldarstellung bei den meisten modernen Mikrocomputern zur externen Wiedergabe der internen Bitgruppen vorgezogen.

DEZIMAL	BINÄR	HEX	OKTAL
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Bild 1-8: Die Hexadezimal Kodes

Selbstverständlich hat die im Speicher des Computers vorliegende Information eine bestimmte Bedeutung; z. B. wird damit ein Text wiedergegeben, oder es sind Zahlen. Zur Wiedergabe dieser Bedeutung für den Menschen ist das Hexadezimalformat jedoch schlecht geeignet.

3. Symbolische Darstellung

Symbolische Darstellung bezieht sich auf die externe Informationswiedergabe mit ihrem symbolischen Bedeutungsinhalt. So werden z.B. Dezimalzahlen als Dezimalzahlen und nicht als Aufeinanderfolge hexadezimaler Zeichen oder einzelner Bits wiedergegeben. Ganz entsprechend wird ein Text als Text und nicht als ASCII-Muster ausgegeben. So ist die symbolische Darstellung die dem Menschen am besten angepaßte Darstellungsform. Man benutzt sie wo immer ein passendes Wiedergabegerät vorhanden ist, wie z.B. ein Bildschirmgerät oder ein Drucker. (Ein Bildschirmgerät, auch Videomonitor genannt, ist eine Einheit mit einem fernseherähnlichen Schirm, auf dem Text oder Grafik dargestellt werden kann.) Leider sind derartige Geräte für kleinere Systeme wie z.B. Einkartencomputer zu teuer, weshalb der Benutzer hier auf den Umgang mit Hexadezimalzahlen eingeschränkt bleibt.

Zusammenfassung zur externen Informationsdarstellung:

Symbolische Darstellung ist immer vorzuziehen, da sie dem menschlichen Benutzer am angepaßtesten ist. Sie benötigt jedoch ein recht teures Interface in Form einer alphanumerischen Tastatur plus Drucker oder Bildschirmgerät. Aus diesem Grund ist sie bei vielen billigeren Systemen nicht ohne weiteres verfügbar. Man benutzt hier alternative Darstellungen, wobei das Hexadezimalformat überwiegt. Nur in den seltenen Fällen, in denen man Detailfehler in Hard- oder Software ansteuern muß, wird auch einmal die Binärdarstellung verwendet. In diesem *Binärformat* werden die Speicher- oder Registerinhalte unmittelbar Bit für Bit dargestellt.

(Über den Sinn direkter Anzeigen auf einer Konsole gibt es seit jeher hitzige Auseinandersetzungen, in die wir uns hier allerdings nicht einmischen wollen.)
Wir haben gerade gesehen, wie man Information intern und extern darstellen kann. Im Folgenden wollen wir den Mikroprozessor untersuchen, der mit dieser Information umzugehen hat.

Zusatzübungen:

Übung 1.28:
Was ist der Vorzug von Zweierkomplementzahlen vor anderen Darstellungsformen vorzeichenbehafteter Zahlen?

Übung 1.29:
Wie würden Sie die Zahl „1024“ in unmittelbar binärer, „Signed-Binary“- und in Zweierkomplementform darstellen?

Übung 1.30:
Was versteht man unter dem V-Bit? Sollte der Programmierer es nach einer Addition oder Subtraktion testen?

Übung 1.31:
Berechnen Sie das Zweierkomplement von „+16", „+17", „+18", „−16", „−17" und „−18".

Übung 1.32:
Schreiben Sie die hexadezimale Form des folgenden im Speicher im ASCII-Format ohne Parität festgehaltenen Textes auf: MELDUNG

KAPITEL 2
HARDWAREORGANISATION DES 6502

Einführung

Beschränkt man sich beim Programmieren auf eine elementare Ebene, so ist es nicht unbedingt notwendig, die interne Struktur des Prozessors bis in alle Einzelheiten zu verstehen. Um jedoch effektiv programmieren zu können, ist ein derartiges Verständnis unbedingt notwendig. Der Zweck dieses Kapitels hier ist die Darstellung der für das Verständnis der Arbeit eines 6502-Systems notwendigen Hardwaregrundlagen. Ein vollständiges Mikrocomputersystem umfaßt nicht nur den Mikroprozessor (hier einen 6502), sondern auch noch andere Bauteile. In diesem Kapitel hier wird der 6502-Aufbau dargestellt, während die anderen Einheiten (in erster Linie solche zur Ein- und Ausgabe) in einem getrennten Kapitel vorgestellt werden (Kapitel 7).
Hier werden wir die Grundarchitektur eines Mikrocomputersystems untersuchen und uns dann dem näheren Studium der internen Organisation des 6502 zuwenden. Insbesondere wollen wir die verschiedenen Register betrachten. Darauf untersuchen wir die Abarbeitung eines Programms. Vom Hardwarestandpunkt aus ist dieses Kapitel jedoch nur eine vereinfachte Darstellung. Der an mehr Details interessierte Leser sei auf unser Buch C201 („From Chips to Systems", vom selben Autor) hingewiesen.

Systemarchitektur

Die Architektur eines Mikrocomputersystems findet sich in Bild 2-1. Auf der linken Seite der Darstellung steht die Mikroprozessoreinheit MPU (microprocessor unit), in unserem Fall ein 6502. Sie führt auf einem Chip die Funktionen einer *Zentraleinheit* (CPU, central processing unit) durch, wozu sie eine *arithmetisch-logische Einheit* (ALU, arithmetic-logical-unit) mit zugeordneten Registern sowie eine *Steuereinheit* (CU, control unit) zur Gewährleistung der richtigen Systemarbeit einsetzt. Ihre Operation wird später genauer beschrieben werden.
Die MPU steht mit dem restlichen System über drei *Busse* in Verbindung: Einen bidirektionalen 8-Bit-Datenbus, der am Kopf der Abbildung steht, einen undirektionalen 16-Bit-Adreßbus und einen Steuerbus am Fuß der Darstellung. Sehen wir uns die Funktion jedes dieser Busse näher an.

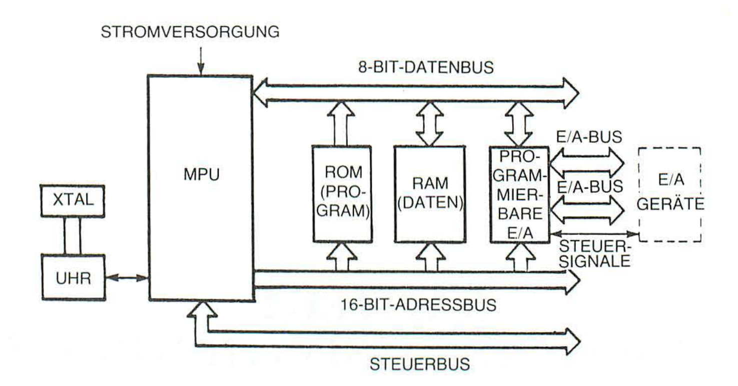


Bild 2-1: Standardarchitektur eines 8-Bit-Mikroprozessorsystems

Der *Datenbus* überträgt die zwischen den verschiedenen Systemeinheiten auszutauschenden *Daten.* In der Regel transportiert er Daten vom Speicher zur MPU, von der MPU zum Speicher oder von der MPU zu einem Ein/Ausgabechip. (Ein Ein/Ausgabechip ist ein Bauteil mit der Aufgabe, Verbindung zu einer externen Einheit aufzunehmen.)

Der *Adreßbus* transportiert *Adressen,* die von der MPU erzeugt wurden und mit denen eines der internen Register in den zum System zusammengeschalteten Chips ausgewählt wird. Diese Adressen legen Quelle oder Ziel der über den Datenbus laufenden Daten fest.

Der *Steuerbus* überträgt die verschiedenen vom System benötigten Synchronisationssignale.

Nachdem wir die Aufgabe der Busse kennengelernt haben, wollen wir noch weitere Bauelemente zur Vervollständigung des Systems anschließen.

Jede MPU benötigt eine genaue Zeitbasis, die von einem *Taktgenerator* zusammen mit einem *Schwingquarz* erzeugt wird. In den meisten „älteren" Mikroprozessoren ist der Taktgenerator außerhalb der MPU auf einem Extrachip untergebracht. Die neueren Mikroprozessoren haben die Taktgeneratorschaltung bereits mit auf dem MPU-Chip integriert. Jedoch muß der Schwingquarz wegen seiner Größe immer noch außerhalb angeschlossen werden. Quarz und Taktgenerator befinden sich in der Zeichnung links von der MPU.

Richten wir unsere Aufmerksamkeit nun auf die anderen Systemelemente. Von links nach rechts lassen sich in der Illustration unterscheiden:

Das *ROM* ist ein *Nur-Lese-Speicher* (read-only-memory) und enthält *Programme* für das System. Der Vorzug eines ROM liegt darin, daß sein Inhalt auch beim Abschalten der Stromversorgung erhalten bleibt. Aus diesem Grund enthält der ROM-Bereich immer ein sogenanntes *Bootstrap-* und ein *Monitorprogramm* (ihre Funktion wird später erklärt werden), die die Operation des Systems nach dem Einschalten gewährleisten. Beim Einsatz zur Prozeßsteuerung stehen nahezu alle Programme im ROM, da sie ziemlich sicher nie geändert werden müssen. Dagegen muß ein Benutzer aus der Industrie sein System gegen Stromversorgungsausfälle schützen: Die Programme dürfen nicht verlorengehen. Sie müssen im ROM stehen.

Im Hobbybereich dagegen oder für ein Programmentwicklungssystem (mit dem der Programmierer das Programm austestet) stehen die meisten Programme im RAM-Bereich, wo sie einfach geändert werden können. Nach Fertigstellung können sie im RAM-Bereich stehen bleiben oder bei Bedarf in ein ROM übertragen werden. RAM-Speicher sind jedoch flüchtig (volatile): Sie verlieren ihren Inhalt, wenn die Stromversorgung abgeschaltet wird.
Der *RAM-Speicher* (von „random access memory" – Speicher mit wahlfreiem Zugriff) ist der *Schreib-Lese-Speicher* des Systems. Im Fall einer Prozeßsteuerung ist der RAM-Bereich üblicherweise recht klein (nur für die zu verarbeitenden Daten). Ein Entwicklungssystem auf der anderen Seite enthält einen großen RAM-Anteil, da es Programme und Software zur Programmerstellung aufnehmen muß. Alle RAM-Inhalte müssen vor ihrer Verwendung erst von einer externen Systemeinheit geladen werden.

Schließlich enthält das System noch einen oder mehrere Interfacebausteine, mit deren Hilfe es mit der Außenwelt verkehren kann. Am häufigsten wird ein sogenanntes *„PIO"*, ein „parallel-input-output"-Chip (d. h. ein Baustein zur parallelen Ein- und Ausgabe) eingesetzt. So ein PIO steht wie alle anderen Chips im System mit allen drei Bussen in Verbindung und stellt mindestens zwei *8-Bit-Tore* (ports) zum Verkehr mit der Außenwelt zur Verfügung. Für weitere Einzelheiten zur Arbeit eines PIO können Sie sich auf das Buch C201 (Chips to Systems) oder ähnliche beziehen; die Besonderheiten eines 6502-Systems finden Sie in Kapitel 7 (Ein/Ausgabe-Einheiten).
Alle diese Chips sind an alle drei Busse einschließlich des Steuerbusses angeschlossen. Um die Abbildung übersichtlicher zu halten, sind die Verbindungen zwischen Steuerbus und den verschiedenen Chips in der Darstellung nicht gezeichnet.
Die eben beschriebenen Funktionseinheiten müssen nicht unbedingt auf separaten LSI-Chips untergebracht werden. Im Gegenteil benutzt man oft *Kombinationsbausteine,* die sowohl ein PIO als auch einen beschränkten ROM- oder RAM-Bereich enthalten. Einzelheiten dazu stehen in Kapitel 7.
Um ein vollständiges System aufzubauen, werden noch weitere Bauteile benötigt. Insbesondere müssen die Busse *gepuffert* werden. Außerdem braucht man *Dekodierlogik* für die RAM-Chips im Speicher, und schließlich müssen einige Signale durch *Treiber* verstärkt werden. Diese Hilfsschaltungen sollen hier nicht weiter beschrieben werden, da sie keinen unmittelbaren Bezug zur Programmierung haben. Der an den Techniken zum Aufbau eines Systems interessierte Leser sei auf unser Buch „Mikroprozessor Interface Techniken" verwiesen.

Die interne Organisation des 6502

Eine vereinfachte Darstellung der internen Organisation des 6502-Prozessors findet sich in Bild 2-2.
In der rechten Bildhälfte steht die arithmetisch-logische Einheit ALU. Sie ist leicht durch ihre charakteristische V-Form auszumachen. Die ALU hat die Aufgabe, mit den ihr an den beiden Eingängen übermittelten Daten arithmetische und logische Operationen auszuführen. Die beiden Eingangstore in die ALU seien als „linker" bzw. „rechter Eingang" bezeichnet. Sie entsprechen den beiden Gabelspitzen des

„V". Nach Beenden einer arithmetischen Operation, wie einer Addition oder einer Subtraktion, gibt die ALU das Ergebnis am unteren Ende in der Zeichnung aus.

Der ALU ist ein spezielles Register, der *Akkumulator* zugeordnet. Dieser Akkumulator ist an den rechten Eingang angeschlossen. Die ALU bezieht sich automatisch auf den Akkumulator als einen ihrer Eingänge. (Es gibt allerdings eine Möglichkeit, das zu umgehen.) Man hat es hier mit einem klassischen akkumulatororientierten Entwurf zu tun. In arithmetischen und logischen Operationen steht einer der Operanden im Akkumulator und der andere üblicherweise irgendwo im Speicher. Das Ergebnis wird wieder im Akkumulator abgelegt. Dieser Bezug auf den Akkumulator sowohl als Datenquelle als auch als Datenziel ist der Grund für dessen Bezeichnung: Er akkumuliert die Ergebnisse. Der Vorteil eines akkumulatororientierten Entwurfs ist die Möglichkeit, sehr kurze Befehle zu verwenden, die gerade ein einziges Byte (8 Bits) zur Festlegung des „Opkodes", d.h. der Art der auszuführenden Operation, benötigen. Wenn der betreffende Operand aus einem anderen Register als dem Akkumulator geholt werden müßte, so benötigt man einige Bits extra, um im Befehl dieses Register bezeichnen zu können. Aus diesem Grund ermöglicht die Akkumulatorarchitektur eine vergrößerte Arbeitsgeschwindigkeit. Der Nachteil ist, daß der Akkumulator vor der betreffenden Operation erst mit den gewünschten Daten geladen werden muß. Das wiederum verschlechtert die Effektivität etwas.
Gehen wir zur Zeichnung zurück. Links von der ALU befinden sich in einem besonderen 8-Bit-Register acht Statusbits, die *Prozessor-Statusflaggen P*. Jedes dieser, durch Flipflops implementierten Bits dient zur Signalisierung eines bestimmten Prozessorzustandes. Die Funktion der verschiedenen Statusbits werden im Lauf der im nächsten Kapitel erarbeiteten Programmbeispiele erläutert und in Kapitel 4 vollständig beschrieben werden, wo der ganze Befehlssatz vorgestellt wird. So gehören zum Beispiel die mit N, Z und C bezeichneten Bits zu den Statusflaggen.
N steht für „negativ". Es ist Bit 7 (das ganz links stehende Bit) in Register P. Immer wenn dieses Bit auf 1 steht, gibt es an, daß das von der ALU ermittelte Ergebnis negativ ist.
Bit Z steht für „zero", d.h. Null. Immer wenn dieses Bit (in Bitstelle 1 des P-Registers) auf 1 steht, wird angezeigt, daß die letzte Rechnung Null erbracht hat.
Bit C, im P-Register ganz rechts auf Position 0 stehend, ist das Übertragsbit (carry). Wenn immer zwei Zahlen addiert werden und das Ergebnis nicht in 8 Bits paßt, enthält C das neunte Bit des Ergebnisses. Der Übertrag wird in arithmetischen Operationen intensiv genutzt.

Diese Statusbits werden von den in Frage kommenden Befehlen automatisch gesetzt oder gelöscht. Eine vollständige Befehlsliste findet sich zusammen mit der Art ihrer Statusbeeinflussung in Anhang A und in Kapitel 4. Diese Bits dienen dem Programmierer zum Test auf besondere Zustände des Systems oder zur raschen Feststellung, ob Fehler aufgetreten sind. Zum Beispiel kann man das Z-Bit mit Hilfe besonderer Befehle testen und erfährt so unmittelbar, ob eine vorangegange Operation das Ergebnis Null (zero) gebracht hat. Alle in Assemblersprache ausgeführten Entscheidungen, d.h. alle in diesem Buch erstellten Programme, basieren auf dem Testen von Bits. Diese Bits werden entweder von außerhalb des Systems geholt oder sind Teil des Statusregisters der ALU. Es ist daher sehr wichtig, Funktion und Einsatz aller Statusbits im System zu verstehen. Die ALU hier enthält ein Statusregister, in dem diese

Bits stehen. Alle anderen Ein/Ausgabechips verfügen ebenfalls über Statusbits. Diese werden in Kapitel 7 behandelt werden.
Gehen wir in Bild 2-2 weiter von der ALU nach links. Die horizontalen Rechtecke dort stehen für die verschiedenen 6502-Register.

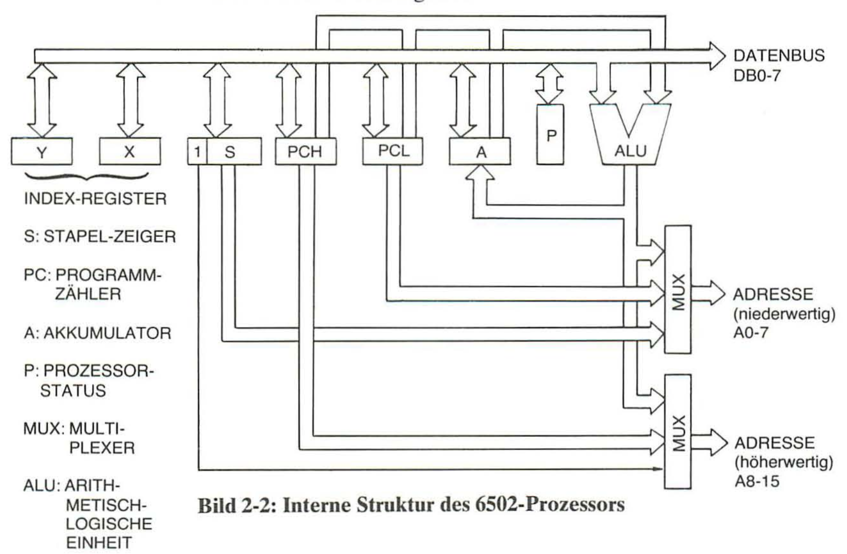


Bild 2-2: Interne Struktur des 6502-Prozessors

PC ist der *Programmzähler* (program counter). Es ist ein 16-Bit-Register und physisch aus zwei Achtbitregistern, PCH und PCL, zusammengesetzt. PLC bezeichnet die niederwertige (low) Hälfte des Programmzählers, d.h. Bit 0 bis 7. PCH steht entsprechend für die höherwertige (high) Hälfte, Bits 8 bis 15. Der Programmzähler ist ein 16-Bit-Register, das die Adresse des nächsten abzuarbeitenden Befehls enthält. Jeder Computer ist mit einem Programmzähler ausgerüstet, durch den er weiß, welcher Befehl als nächster an die Reihe kommt. Sehen wir uns zum genaueren Verständnis der Rolle des Programmzählers kurz den Mechanismus des Speicherzugriffs im System an.

Der Befehlszyklus

Betrachten wir Bild 2-3. Auf der linken Seite befindet sich die MPU und rechts davon der Speicher. Es kann sich dabei um ROM, RAM oder sonst einen mit Speichern versehenen Chip handeln. Dieser Speicher dient dazu, Befehle und Daten festzuhalten. In unserem Fall wollen wir aus dem Speicher einen Befehl übernehmen, um die Aufgabe des Programmzählers zu verdeutlichen. Nehmen wir dazu an, daß der Programmzähler einen sinnvollen Inhalt besitzt. Er enthält dann eine 16-Bit-Adresse, die den Ort des nächsten aus dem Speicher zu holenden Befehls bezeichnet. Jeder Prozessor arbeitet dabei in drei Schritten:

1 – den nächsten Befehl holen,
2 – den Befehl dekodieren,
3 – den Befehl ausführen.

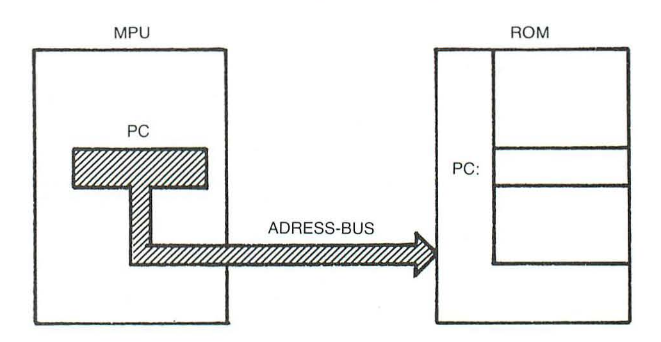


Bild 2-3: Befehlsübernahme aus dem Speicher

Befehl holen

Sehen wir uns einen solchen Befehlszyklus einmal an. Im ersten Schritt wird der Inhalt des Programmzählers auf den Adreßbus gelegt und an den Speicher weitergegeben. Gleichzeitig wird (falls notwendig) über den Steuerbus des Systems ein Lesesignal ausgegeben. Der Speicher übernimmt die Adresse, die eine bestimmte unter den Speicherstellen anfordert. Wenn Lesesignal ankommt, dekodiert der Speicher die erhaltene Adresse mit Hilfe interner Dekodierer und aktiviert die angeforderte Stelle. Ein paar hundert Nanosekunden später legt der Speicher den in der adressierten Stelle vorgefundenen Inhalt auf den Datenbus. Dieses Achtbitwort enthält den benötigten Befehl.
Fassen wir das kurz zusammen. Der Programmzählerinhalt wird auf den Datenbus gelegt. Ein Lesesignal wird erzeugt. Der Speicher arbeitet. Ungefähr 300 Nanosekunden darauf wird der an der betreffenden Adresse stehende Befehl auf den Datenbus gelegt. Der Mikroprozessor liest diesen Datenbusinhalt und legt ihn in einem speziellen internen Register, dem *Befehlsregister* IR (instruction register) ab. Es ist acht Bits breit und soll den jeweils zuletzt übernommenen Befehl festhalten. Damit ist die Befehlsübernahme erledigt. Die acht Befehlsbits stehen nun in einem besonderen internen 6502-Register. Dieses IR-Register befindet sich links im Bild 2-4.

Dekodierung und Ausführung

Wenn der Befehl in das IR-Register übernommen ist, wird er von der Steuereinheit im Mikroprozessor entschlüsselt, die daraufhin eine Folge von internen und externen Signalen zur Befehlsausführung erzeugt. Zu diesem Zweck wird etwas Zeit benötigt, weshalb der Befehlsübernahme eine kurze Dekodierverzögerung folgt, deren Länge von der Art des Befehls abhängt. Einige Befehle werden vollständig im Innern der MPU abgearbeitet. Andere Befehle holen weitere Daten aus dem Speicher oder legen solche dort ab. Aus diesem Grund brauchen die verschiedenen 6502-Befehle unterschiedlich viel Zeit zur Ausführung. Diese Zeit wird als Anzahl von Taktzyklen angegeben, der für jeden Befehl in der Zusammenstellung in Anhang A aufgeführt sind.

Ein typischer 6502-Prozessor arbeitet bei einer Taktfrequenz von 1 MHz. Damit beträgt die Länge eines solchen Taktzyklus eine Mikrosekunde. Da verschiedene Bauteile andere Taktfrequenzen ermöglichen, gibt man die Dauer der Befehlsabarbeitung lieber in Taktzyklen als in Nanosekunden an.
Im Fall des 6502 wird der Takt intern durch einen Taktgenerator auf dem Chip erzeugt. (vgl. Bild 2-1).

Übernahme des nächsten Befehls

Wir haben damit beschrieben, wie man mit Hilfe des Programmzählers einen Befehl aus dem Speicher holen kann. Zur Programmabarbeitung müssen diese Befehle *einer nach dem anderen* übernommen werden. Damit brauchen wir einen automatischen Mechanismus, der dieses Nacheinander gewährleistet. Diese Aufgabe wird durch einen einfachen Inkrementierer, eine Weiterzählschaltung beim Programmzähler erledigt. In Bild 2-4 ist das dargestellt. Jedesmal wenn der Inhalt des (am Fuß der Zeichnung stehenden) Programmzählers auf den Adreßbus gelegt wird, wird er um 1 weitergezählt und in den Programmzähler zurückgeschrieben. Wenn der Programmzähler z.B. die Adresse 2304 enthält, so wird 2304 auf den Adreßbus ausgegeben. Darauf wird dieser Wert vom Inkrementierer auf 2305 weitergezählt und in den Programmzähler zurückübertragen. Auf diese Weise wird beim nächsten Befehlszyklus der Befehl von Adresse 2305 übernommen. Wir haben damit einen automatischen Mechanismus zur schrittweisen Befehlsabarbeitung gefunden.
Es muß allerdings betont werden, daß diese Beschreibung vereinfacht ist. In der Praxis gibt es Befehle, die zwei oder gar drei Bytes umfassen, die nacheinander in den Prozessor geholt werden müssen. Der Grundmechanismus ist jedoch derselbe.

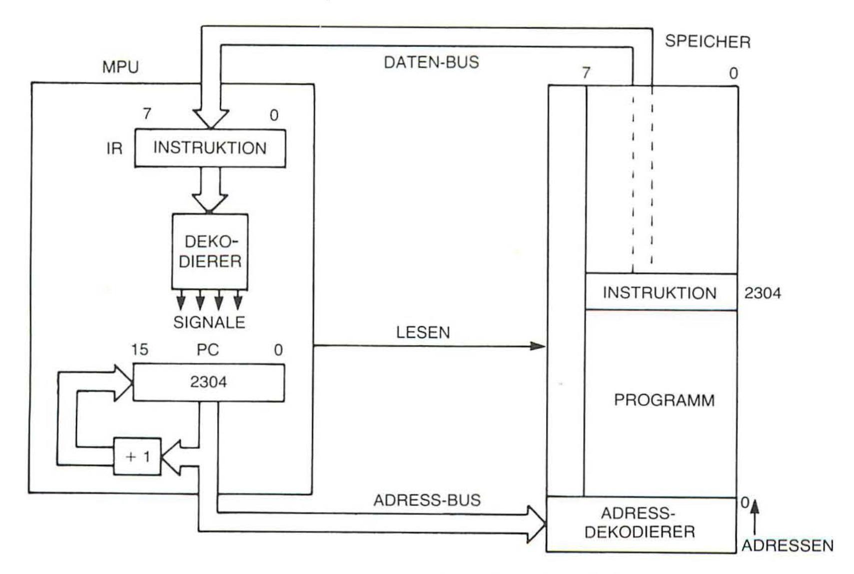


Bild 2-4: Der Programmzähler wird bei der Befehlsübernahme automatisch weitergezählt

Mit Hilfe des Programmzählers holt man ein Befehlsbyte nach dem anderen genauso aus dem Speicher wie man einen Einbytebefehl nach dem anderen übernimmt. Zusammen mit dem Inkrementierer kann man über den Programmzähler aufeinanderfolgende Speicherstellen erreichen.

Weitere 6502-Register

Bei unserer Untersuchung von Bild 2-2 fehlt noch ein Bereich. Er umfaßt einen Satz von drei Registern, X, Y und S. Register X und Y werden als *Indexregister* bezeichnet. Sie sind 8 Bits breit und können Daten aufnehmen, mit denen das Programm arbeitet. Ihr Hauptzweck allerdings besteht in ihrem Einsatz bei der Adressierung.
Die Aufgabe von Indexregistern wird in Kapitel 5 bei der Darstellung der Adressierungstechniken genauer beschrieben werden. Kurz gesagt, kann man den Inhalt dieser Register zu jeder Adresse im System addieren und so einen automatischen Versatz, einen „Offset" erhalten. Das ist eine wichtige Eigenschaft für den effektiven Zugriff auf in Tabellen festgehaltene Daten. Die beiden Register X und Y besitzen allerdings nicht genau dieselben Eigenschaften und haben daher etwas verschiedene Funktion, was in Kapitel 7 genauer auseinandergesetzt wird.
Der *Stapelzeiger* S (Stackpointer) dient zur Aufnahme eines Zeigers auf die Spitze des im Speicher liegenden Speicherbereichs.
Hierfür sollten wir uns das Stapelkonzept ansehen.

Der Stapel (Stack)

Man bezeichnet den Stapel nach seiner Funktion auch als LIFO-Struktur (last-in, first-out: das zuletzt Eingegebene wird zuerst wieder ausgegeben). Ein solcher Stapel besteht aus einer Anzahl von Registern oder Speicherstellen, die diese Struktur nachbilden. Die wesentliche Eigenschaft dieser Struktur ist ihre *chronologische* Arbeit. Das zuallererst auf den Stapel gebrachte Element befindet sich immer auf dessen Boden, das zuletzt eingebrachte dagegen immer auf der Stapelspitze. Man kann das mit einer Art von Groschenspeicher vergleichen. Diese bestehen im Prinzip aus einem Zylinder, der eine Schraubenfeder enthält und in den man durch einen Schlitz Parkgroschen oder Entsprechendes auf die Feder stecken kann. Jeder neu eingesteckte Groschen schiebt die vorigen eine Stelle nach unten in den Zylinder hinein, so daß der zuerst eingegebene sich immer ganz unten befindet. Der zuletzt eingeschobene Groschen dagegen liegt auf der Spitze und kann bei Bedarf (als erster) aus dem Schlitz entnommen werden. Dadurch wird zugleich auch eine andere Eigenschaft deutlich. Üblicherweise genügen zum Zugriff auf seinen Inhalt zwei Befehle: *Auf den Stapel schieben* („push") und *vom Stapel herunterziehen* („pull" oder „pop"). Ein PUSH-Befehl bewirkt, daß ein Element auf der Stapelspitze abgelegt wird. Ein PULL-Befehl ermöglicht dagegen das Herunternehmen eines Elements von der Stapelspitze. In der Mikroprozessorpraxis wird normalerweise der *Akkumulatorinhalt* auf die Stapelspitze gebracht bzw. mit PULL oder POP (je nach Prozessortyp) der Inhalt der Stapelspitze in den Akkumulator zurückübertragen. Es gibt auch oft Befehle, mit denen der Inhalt anderer Prozessorregister, in erster Linie der des Statusregisters auf den Stapel gebracht bzw. von dort geladen werden kann.

Man braucht einen Stapel zur Implementation dreier Programmiermöglichkeiten in einem Computersystem: Unterprogramme (Subroutinen), Programmunterbrechungen (Interrupts) und vorübergehende Datenspeicherung. Die Rolle des Stapels bei Unterprogrammen wird in Kapitel 3 (Grundlegende Programmiertechniken) beschrieben. Die Aufgabe des Stapels bei Programmunterbrechungen ist in Kapitel 6 erklärt. Und die Funktion des Stapels als schneller Zwischenspeicher wird im Zuge der verschiedenen Programmbeispiele angesprochen werden.

Wir wollen an dieser Stelle einfach annehmen, daß man für die Arbeit des Computersystems einen Stapel benötigt. Ein solcher Stapel kann auf zwei verschiedene Arten implementiert werden:

1. Im Mikroprozessor selbst wird eine Gruppe von Registern bereitgestellt. So etwas nennt man einen „Hardwarestapel". Er hat den Vorteil hoher Arbeitsgeschwindigkeit. Nachteilig ist allerdings, daß die dazu zur Verfügung stehende Registerzahl beschränkt ist.

2. Die meisten Allzweckmikroprozessoren verwenden einen anderen Ansatz, den „Softwarestapel", um die Stapelgröße nicht durch eine kleine Registeranzahl einzuschränken. Dieser Ansatz ist auch im 6502-Entwurf übernommen wurden. Im Softwareansatz ist im Prozessor ein spezielles Register, der Stapelzeiger S, vorgesehen. Dieser enthält immer die Adresse des Elements auf der Stapelspitze (genauer gesagt: die Spitzenadresse plus Eins). Der Stapel selbst wird in einem passenden Speicherbereich untergebracht. Damit braucht der Stapelzeiger einen Umfang von 16 Bits, um jede Speicherstelle erreichen zu können.

Beim 6502 hat man den Stapelzeiger jedoch auf 8 Bit eingeschränkt. Dem Register ist ein neuntes Bit ganz links zugeordnet, das immer den Wert 1 hat. Mit anderen Worten: Der für den 6502-Stapel verfügbare Speicherbereich reicht von Adresse 256 bis Adresse 511, binär „100000000" bis „111111111". Dabei beginnt der Stapel immer mit Adresse „111111111" und kann bis zu 256 Worte umfassen. Dieser eingeschränkte Stapelbereich ist eine 6502-typische Einschränkung und soll später im Buch genauer untersucht werden. Im 6502 beginnt der Stapel des weiteren mit der größten Adresse und wächst „rückwärts". Der Stapelzeiger wird durch eine PUSH-Operation dekrementiert, d.h. um 1 heruntergezählt.

Um den Stapel benutzen zu können, braucht man den Inhalt des S-Registers nur auf einen bestimmten Anfangswert zu setzen, d.h. zu initialisieren. Der Rest läuft automatisch ab.

Man sagt, der Stapel liege auf *Speicherseite* (Page) 1, das bringt uns zum *Seitenkonzept* bei der Speichereinteilung.

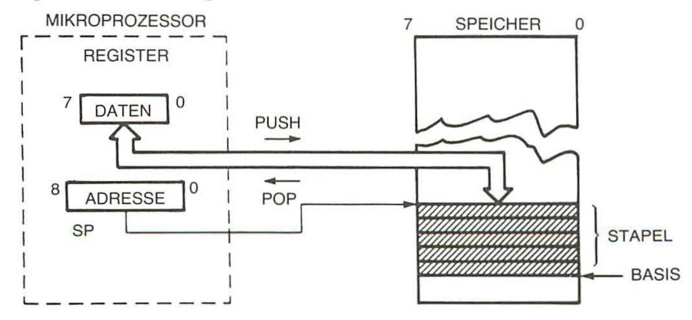


Bild 2-5: Die zwei Befehle zum Umgang mit Stapelspeichern

Das Konzept der Speicherseiten

Der 6502-Prozessor besitzt einen 16-Bit-Adreßbus. 16 Bits ermöglichen die Adressierung von $2^{16} = 64$ K Speicherstellen (1 K entspricht dem Wert 1024). Wegen der in Kapitel 5 dargelegten Adressierbesonderheiten des 6502-Prozessors teilt man den Speicher in mehrere logische *Seiten* ein. Eine Speicherseite ist nichts weiter als ein zusammenhängender Block von 256 Speicherzellen. Z.B. liegen die Speicherzellen mit den Adressen 0 bis 255 auf Seite 0 (page 0) des Speichers. Diese Seite hat eine besondere Bedeutung bei der Adressierung. Man benutzt sie im sogenannten „zero-page"-Modus. Speicherseite 1 umfaßt die Adressen 256 bis 511. Wir haben eben festgestellt, daß dies der für den Stapel reservierte Bereich ist. Alle anderen Speicherseiten unterliegen keinen besonderen Bedingungen und können beliebig eingesetzt werden. Es ist beim 6502 allerdings wichtig, sich diese Seiteneinteilung zu merken. In vielen Fällen verlangsamt sich die Befehlsabarbeitungszeit um einen zusätzlichen Taktzyklus, wenn dabei eine Seitengrenze überschritten wird.

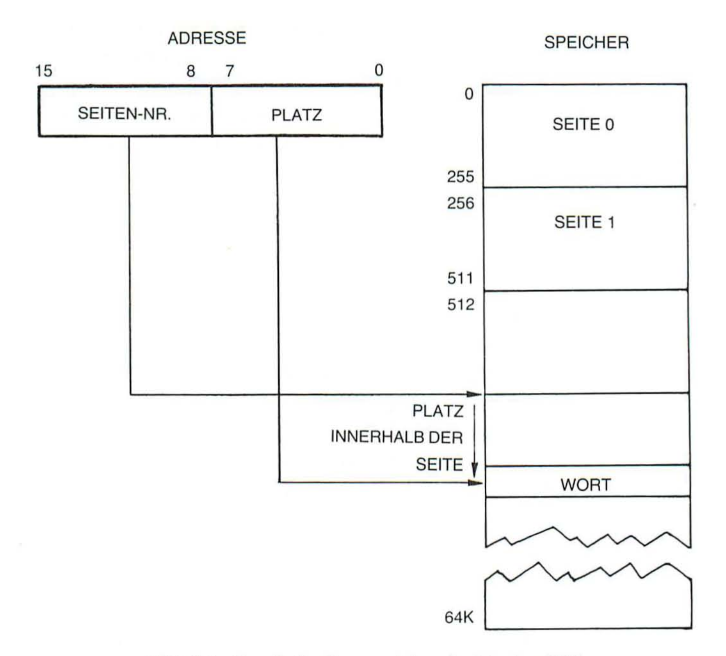


Bild 2-6: Das Seitenkonzept (paging) beim 6502

Der 6502-Chip

Der Vollständigkeit halber sei gesagt, daß der oben in Bild 2-2 erscheinende Datenbus den externen Datenbus wiedergibt. Man benutzt ihn zum Verkehr mit externen Systemeinheiten, in erster Linie mit dem Speicher. A0 bis A7 und A8 bis A15 bezeichnen die nieder- und höherwertigen Hälften des 6502-Adreßbusses.

Des weiteren wollen wir unsere 6502-Betrachtung mit der Anschlußbelegung des Chips komplettieren. Dieser Teil ist zum Verständnis des restlichen Buchinhalts nicht nötig. Sie können diesen Abschnitt überspringen, wenn Sie wollen. Wenn Sie jedoch weitere Einheiten an Ihr System anschließen möchten, mag die folgende Darstellung hilfreich sein. Die Anschlußbelegung des 6502-Prozessorchips zeigt Bild 2-7. Der Datenbus ist mit DB0-7 gekennzeichnet und einfach auf der rechten Bildseite aufzufinden. Der Adreßbus ist mit A0-11 und A12-15 bezeichnet. Er entspricht auf der rechten Chipseite aus Anschluß 9 bis 20 und auf der linken aus Anschluß 22 bis 25.
Der Rest umfaßt die Stromversorgung und die Steuerleitungen.

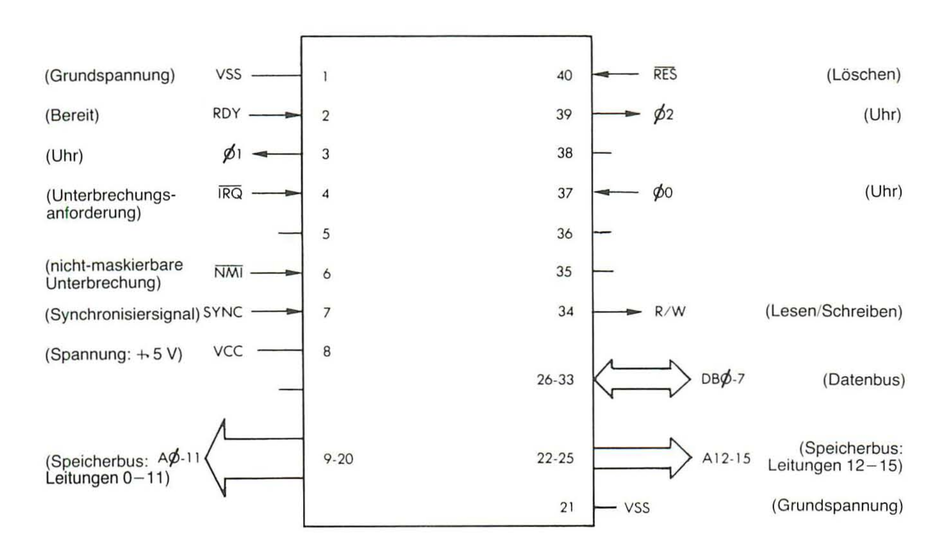


Bild 2-7: Anschlußbelegung des 6502-Prozessors

Die Steuerleitungen

- R/W: Diese Schreib/Lese-Leitung READ/WRITE bestimmt die Richtung, in der die Daten auf dem Datenbus übertragen werden.
- IRQ und NMI stehen für die Unterbrechungsanforderungen „interrupt request“ und NMI „non maskable interrupt“ (nicht maskierbare Unterbrechung).
- SYNC ist ein Synchronisiersignal, mit dem die Übernahme eines Befehlsbytes dem restlichen System angezeigt wird.
- RDY (ready) dient üblicherweise zur Synchronisation der Prozessorarbeit mit langsamen Speicherbausteinen: Er stoppt bei Bedarf den Prozessor.
- SO (set overflow flag) setzt die Überlaufflagge von außen. Dieser Eingang wird üblicherweise nicht benutzt.
- $\varnothing_0$, $\varnothing_1$ und $\varnothing_2$ sind Taktsignale.
- V_{SS} und V∞ dienen zur Stromversorgung (mit 5 V).

Zusammenfassung zur Hardware

Damit ist unsere Beschreibung der 6502-Hardware vollständig. Die genaue Struktur der internen Busse ist an dieser Stelle nicht wichtig. Jedoch ist die genaue Funktion jedes Registers wesentlich und sollte vor Weitergehen im Buch vollständig verstanden sein. Wenn Sie sich mit den vorgestellten Konzepten vertraut fühlen, dann können Sie weiterlesen. Sollten Sie aber bei dem einen oder anderen Punkt unsicher sein, so sollten Sie unbedingt die wichtigsten Stellen dieses Kapitels noch einmal lesen, da sie im folgenden ständig gebraucht werden. Sehen Sie dazu noch einmal Bild 2-2 an und stellen Sie sicher, daß Sie die Funktion jedes Registers dort verstanden haben.

KAPITEL 3
GRUNDLEGENDE PROGRAMMIERTECHNIKEN

Einführung

Ziel dieses Kapitels ist die Darstellung aller zum Programmieren des 6502 notwendigen grundlegenden Programmiertechniken. Außerdem werden weiterführende Konzepte wie Registerverwaltung, Programmschleifen und Unterprogramme eingeführt. Dabei liegt der Schwerpunkt auf Programmiertechniken, die nur die *internen* Möglichkeiten des 6502, d.h. die Register ausnutzen. Es werden praktisch brauchbare Programme, in erster Linie Arithmetikprogramme entwickelt. Diese Programme dienen zur Illustration der bis dahin entwickelten Konzepte und entsprechen in Aufbau und Befehlsstruktur der Praxis. Man kann daraus sowohl entnehmen, wie man Befehle zur Handhabung des Datenverkehrs zwischen Speicher und MPU einsetzt, als auch wie man mit der Information innerhalb der MPU umgeht. Das nächste Kapitel wird dann die für den 6502 verfügbaren Befehle in allen Einzelheiten behandeln. Kapitel 6 bringt die zum Umgang mit Information *außerhalb* der MPU notwendigen Methoden: die Ein/Ausgabetechniken.
In diesem Kapitel hier soll in erster Linie durch die Praxis gelernt werden. Indem wir Programme mit wachsender Komplexität untersuchen, werden wir die Rolle der verschiedenen Befehle und Register kennenlernen und die bis dahin entwickelten Konzepte anwenden. Ein wichtiges Konzept wird jedoch noch ausgelassen, das der Adressierungstechniken. Wegen seiner Komplexität wird es für sich in Kapitel 5 dargestellt werden.
Fangen wir ohne weitere Umschweife damit an, ein paar Programme für den 6502 zu schreiben. Beginnen wir mit Arithmetikprogrammen.

Arithmetikprogramme

Arithmetikprogramme befassen sich mit Addition, Subtraktion, Multiplikation und Division. Die hier vorgestellten Programme werden mit ganzen Zahlen arbeiten. Diese können rein binär positive oder im Zweierkomplement dargestellte positive und negative Zahlen sein, wobei das ganz links stehende Bit das Vorzeichen angibt. (Wenn Sie sich über die Zweierkomplementdarstellung nicht mehr ganz sicher sind, sollten Sie in Kapitel 1 nachschlagen.)

8-Bit-Addition

Addieren wir zwei mit OP1 und OP2 bezeichnete 8-Bit-Operanden, die unter den Speicheradressen ADR1 bzw. ADR2 festgehalten sein sollen. Die Summe sei RES genannt und im Speicher unter Adresse ADR3 abgelegt. Dies ist in Bild 3-1 verdeutlicht. Das folgende Programm führt diese Addition aus:

```
LDA    ADR1    OP1 IN A HOLEN
ADC    ADR2    OP2 ZU OP1 ADDIEREN
STA    ADR3    RES UNTER ADR3 ABLEGEN
```

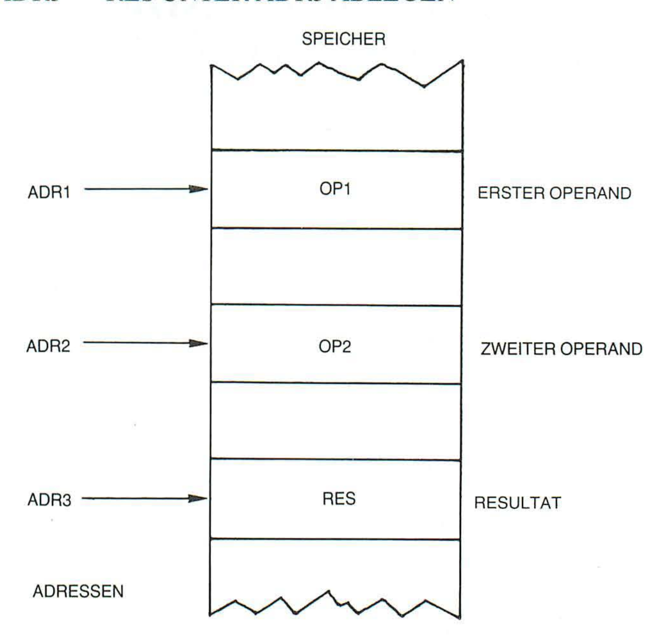


Bild 3-1: 8-Bit-Addition: RES = OP1 + OP2

Das ist ein Programm aus drei Befehlen. In jeder Zeile steht in symbolischer Form ein Befehl. Diese Befehle werden von einem Assemblerprogramm in jeweils ein, zwei oder drei binäre Bytes übersetzt. Wir wollen uns hier allerdings nicht mit dieser Übersetzung beschäftigen und nur die symbolische Wiedergabe betrachten. Die erste Zeile gibt einen LDA-Befehl an. LDA bedeutet „load accumulator A", d.h. „Akkumulator A laden", wobei das zu Ladende von der dem Befehl folgenden Adresse geholt werden soll.

Die in der ersten Zeile angegebene Adresse lautet ADR1. ADR1 steht symbolisch für eine tatsächliche 16-Bit-Adresse. Irgendwoanders im Programm wird ADR1 definiert. Zum Beispiel könnte es sich um Adresse 100 handeln.

Der Befehl LDA besagt damit „Lade den Akkumulator A" (innerhalb des 6502) mit dem Wert unter Speicheradresse 100. Das bewirkt eine Leseoperation von Adresse

100, deren Inhalt über den Datenbus übertragen und im Akkumulator abgelegt wird. Sie werden sich daran erinnern, daß die arithmetischen und die logischen Operationen als einen ihrer Operanden den Akkumulator heranziehen. (Für mehr Einzelheiten schlagen Sie bitte im vorigen Kapitel nach.) Da wir die beiden Werte OP1 und OP2 zusammenzählen möchten, müssen wir zunächst OP1 in den Akkumulator holen. Danach können wir den Akkumulatorinhalt (OP1) zu OP2 addieren.
Das ganz rechts stehende Feld dieser Befehle wird *Kommentarfeld* genannt. Es wird vom Prozessor ignoriert und dient der Lesbarkeit des Programms. Um zu verstehen, was das Programm macht, ist es entscheidend wichtig, gute Kommentare einzusetzen. So etwas nennt man die *Dokumentation* des Programms. Der Kommentar in unserem Fall bedarf keiner weiteren Erläuterung: Der OP1-Wert unter Adresse ADR1 wird in den Akkumulator geladen.
Der Abarbeitungsvorgang dieses ersten Befehls ist in Bild 3-2 verdeutlicht.

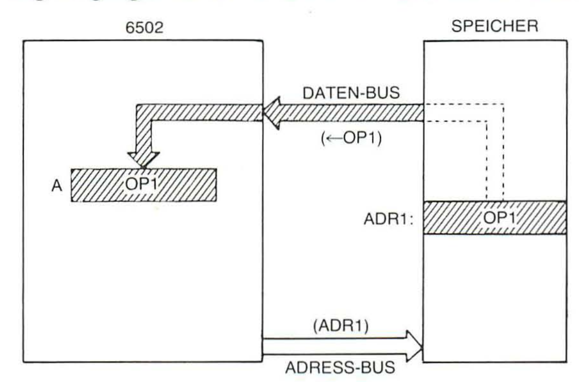


Bild 3-2: LDA ADR1: Der Operand OP1 wird aus dem Speicher in den Akkumulator geholt

Der zweite Befehl unseres Programms lautet:

ADC ADR2

Er besagt „Inhalt der Speicherstelle unter ADR2 zum Akkumulator addieren". Bild 3-1 zeigt, daß in ADR2 unser zweiter Operand OP2 steht. Im Akkumulator haben wir zur Zeit unseren ersten Operanden OP1 stehen. Beim Ausarbeiten des zweiten Befehls wird OP2 aus dem Speicher geholt und zu OP1 addiert. Die Summe wird im Akkumulator abgelegt. Erinnern Sie sich: Beim 6502 wird das Ergebnis einer arithmetischen Operation in den Akkumulator zurück übertragen. Andere Mikroprozessoren können die Möglichkeit bieten, das Ergebnis in anderen Registern oder wieder im Speicher abzulegen.
Damit steht die Summe aus OP1 und OP2 im Akkumulator. Wir müssen nur noch den Akkumulatorinhalt in Speicherstelle ADR3 übertragen, damit das Ergebnis an die verlangte Stelle kommt. Auch hier ist das ganz rechts stehende Feld der zweiten Befehlszeile nichts weiter als ein Kommentar, der die Aufgabe des Befehls beschreibt (OP2 zu A addieren).

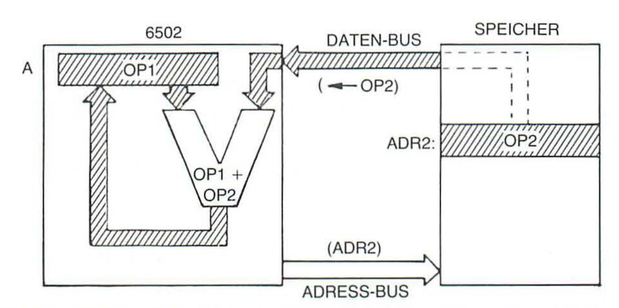


Bild 3-3: ADC ADR2: Operand OP2 wird zum Akkumulatorinhalt addiert und das Ergebnis im Akkumulator festgehalten

Die Wirkung des zweiten Befehls verdeutlicht Bild 3-3.
Anhand von Bild 3-3 läßt sich nachprüfen, daß der Akkumulator zunächst OP1 enthält. Nach der Addition wird das Ergebnis OP1 + OP2 in den Akkumulator geschrieben. Der Inhalt aller anderen Register im System bleibt genauso wie der aller Speicherstellen während einer Leseoperation unverändert. Mit anderen Worten: *Das Auslesen des Inhalts eines Registers oder einer Speicherstelle verändert diesen in keiner Weise.* Ausschließlich Schreiboperationen können den Inhalt eines Registers verändern! In diesem Beispiel bleibt der Inhalt der Speicherstellen ADR1 und ADR2 unverändert. Nach Abarbeiten des zweiten Befehls in unserem Programm jedoch ist der Inhalt des Akkumulators ein anderer, da die ALU-Ausgabe in den Akkumulator eingeschrieben worden ist. Damit geht sein vorheriger Inhalt verloren.
Speichern wir dieses Ergebnis zum Abschluß unserer einfachen Addition noch unter ADR3 ab.
Der dritte Befehl besagt: STA ADR3. STA heißt: „store the contents of accumulator A" – „den Inhalt von Akkumulator A abspeichern". Damit lautet unser Befehl: Speichere den Inhalt von Akkumulator A in der Speicherstelle mit der Adresse ADR3 ab. Das spricht für sich selbst und ist in Bild 3-4 dargestellt.

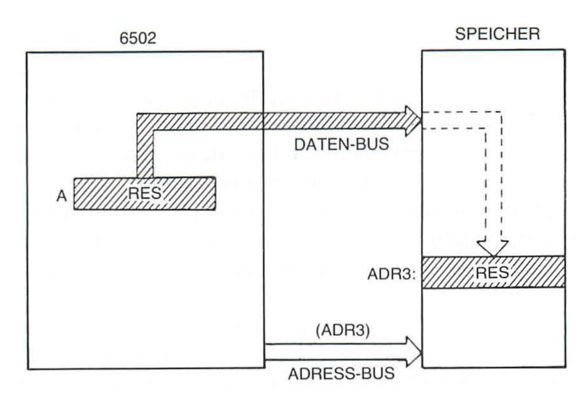


Bild 3-4: STA ADR3: Der Akkumulatorinhalt wird im Speicher unter Adresse ADR3 abgelegt

6502-Besonderheiten

Das Dreibefehlsprogramm von oben wäre für die meisten Mikroprozessoren in der Tat alles, was man braucht. Es gibt jedoch zwei Besonderheiten beim 6502, wegen deren normalerweise zwei weitere Befehle nötig sind.
Zunächst heißt ADC genau betrachtet „add with carry", d.h. mit Übertrag addieren, und nicht nur „add". Der Unterschied liegt darin, daß ein normaler Additionsbefehl nur zwei Zahlen zusammenzählt. Eine Addition mit Übertrag dagegen zählt die beiden Zahlen zusammen und addiert dazu noch den Wert des Übertragsbits. Da wir hier nur zwei 8-Bit-Zahlen addieren, darf kein Übertrag benutzt werden. Andererseits kennt man den Wert des Übertragsbits C in der Regel beim Einleiten der Addition nicht (es kann von einem vorhergehenden Befehl gesetzt worden sein), also müssen wir es erst löschen, d.h. auf Null setzen. Das läßt sich durch den Befehl „Übertrag löschen" (clear carry) erreichen, der im 6502-Befehlssatz CLC heißt.
Leider verfügt der 6502 nicht über beide Möglichkeiten zur Addition. Er kennt nur die ADC-Operation. Das erfordert bei einfachen 8-Bit-Operationen die notwendige Vorsorge, immer erst das Übertragsbit C löschen zu müssen. Das ist kein schwerwiegender Nachteil, darf aber nicht vergessen werden.
Die zweite Besonderheit des 6502 liegt in der Tatsache, daß er mit leistungsfähigen Befehlen zur dezimalen Arithmetik ausgerüstet ist. Wir werden im Abschnitt über BCD-Arithmetik darauf genauer eingehen. Der 6502 arbeitet immer in einer von zwei Betriebsarten: binär oder dezimal. Die jeweilige Betriebsart wird durch das Statusbit „D" (decimal mode - Dezimalbetrieb) in Register P vorgeschrieben. Da wir in diesem Beispiel hier binär arbeiten, ist es wichtig, daß das D-Bit richtig gesetzt ist. Das läßt sich mit dem CLD-Befehl erreichen, der dieses löscht („clear D"). Selbstverständlich reicht es für ein Programm aus, wenn das D-Bit am Programmanfang ein für allemal auf den zugehörigen Wert (0 für binäre, 1 für dezimale Operation) gesetzt wird. Es braucht nicht vor jeder Rechnung neu gesetzt oder gelöscht zu werden. Aber es muß mindestens einmal im Programm definiert werden. Da Sie beim Üben öfter zwischen binärer und dezimaler Arbeitsweise wechseln dürfen, haben wir den zugehörigen Befehl hier mit angegeben:
Fassen wir zusammen: Unser vollständiges und sicheres Programm zur binären 8-Bit-Addition lautet:

```
CLD              DEZIMALMODUS ABSCHALTEN
CLC              ÜBERTRAG LÖSCHEN
LDA     ADR1     OP1 IN A HOLEN
ADC     ADR2     OP2 ZU OP1 ADDIEREN
STA     ADR3     RES UNTER ADR3 ABLEGEN
```

Man könnte statt ADR1, ADR2 oder ADR3 auch die tatsächlichen Adressen angeben. Sollen jedoch die symbolischen Adressen beibehalten werden, so ist es notwendig, ihnen mit Hilfe sogenannter „Pseudobefehle" irgendwo im Programm den tatsächlich gebrachten Wert zuzuweisen. Mit Hilfe dieser Pseudobefehle kann das Assemblerprogramm beim Übersetzen die symbolischen durch ihre tatsächlichen Werte ersetzen.
Derartige Pseudobefehle können z.B. so aussehen:

```
ADR1 = $ 100
ADR2 = $ 120
ADR3 = $ 200
```

Übung 3.1:
Schließen Sie jetzt das Buch. Beziehen Sie sich ausschließlich auf die Befehlszusammenstellung im Anhang. Schreiben Sie ein Programm, das die beiden in den Speicherstellen LOC1 und LOC2 stehenden Zahlen addiert. Legen Sie das Ergebnis in Speicherstelle LOC3 ab. Vergleichen Sie dann Ihr Programm mit dem obenstehenden.

16-Bit-Addition

Mit einer 8-Bit-Addition kann man 8-Bit-Zahlen addieren, d.h. Werte von 0 bis 255, falls reine Dualzahlen verwendet werden. Die meisten praktischen Anwendungen erfordern jedoch den Einsatz von *mehrfacher Genauigkeit,* d.h. Addition von Zahlen mit 16 und mehr Bits Länge. Wir wollen uns hier Beispiele zur 16-Bit-Addition ansehen. Diese können ohne weiteres auf 24, 32 oder mehr Bits ausgedehnt werden. (Dabei kommen immer Mehrfache von 8 Bits zur Anwendung.) Nehmen wir an, der erste Operand sei in den Speicherstellen ADR1 und ADR1-1 abgelegt. Da OP1 diesmal eine 16-Bit-Zahl ist, benötigt man für sie zwei 8-Bit-Speicherstellen. Entsprechend wird OP2 unter ADR2 und ADR2-1 gespeichert. Und das Ergebnis soll nach ADR3 und ADR3-1 übertragen werden. Diese Speicherbelegung ist in Bild 3-5 wiedergegeben.

SPEICHER

ADR1 1	(OP1)H
ADR1	(OP1)L
ADR2 1	(OPR2)H
ADR2	(OPR2)L
ADR3 1	(RES)H
ADR3	(RES)L

Bild 3-5: 16-Bit-Addition: Die Operanden

Die diesem Problem zugrunde liegenden Überlegungen entsprechen genau denen für unser erstes Beispiel. Zunächst werden die niederwertigen Operandenhälften addiert, da der Mikroprozessor immer nur 8 Bits auf einmal verarbeiten kann. Ein bei dieser Addition etwa auftretender Übertrag wird automatisch im Übertragsbit C gespeichert. Darauf werden die beiden höherwertigen Operandenhälften zusammen mit diesem Übertrag addiert und das Ergebnis im Speicher festgehalten werden. Das Programm dazu sieht so aus:

```
CLD
CLC
LDA     ADR1     OP1, NIEDERWERTIGE HÄLFTE
ADC     ADR2     (OP1 + OP2), NIEDERWERTIGE HÄLFTE
STA     ADR3     NIEDERWERTIGE ERGEBNISHÄLFTE SPEICHERN
LDA     ADR1-1   OP1, HÖHERWERTIGE HÄLFTE
ADC     ADR2-1   (OP1 + OP2), HÖHERWERTIGE HÄLFTE + CARRY
STA     ADR3-1   HÖHERWERTIGE ERGEBNISHÄLFTE SPEICHERN
```

Die beiden ersten Befehle, CLD und CLC, sind zur Sicherheit angefügt. Ihre Aufgabe ist im vorigen Abschnitt beschrieben worden. Die drei nächsten Befehle sind dieselben wie vorhin bei der 8-Bit-Addition. Sie bewirken, daß die niederwertigen Operandenhälften (Bits 0 bis 7) addiert werden. Die RES genannte Summe daraus wird in Speicherstelle ADR3 abgelegt.

Immer wenn eine Addition ausgeführt wird, erhält das Übertragsbit C des Statusregisters P den Wert des entstandenen Übertrags. Wird kein Übertrag in die nächste Bytestelle erzeugt, so hat C den Wert Null. Entsteht dagegen ein Übertrag, so wird das C-Bit auf Eins gesetzt.

Die drei folgenden Befehle im Programm entsprechen wieder genau denen der 8-Bit-Addition. Mit ihnen werden die höherwertigen Operandenhälften (Bits 8 bis 15) plus Übertrag C addiert und die so entstehende höherwertige Ergebnishälfte in ADR3-1 abgelegt. Auf diese Weise steht nach Abarbeiten des Programms in den Speicheradressen ADR3 und ADR3-1 das 16-Bit-Ergebnis der Addition.

Wir nehmen an, daß kein Übertrag aus dieser 16-Bit-Addition heraus entsteht, d.h. daß das Ergebnis in 16 Bits unterzubringen ist. Wenn beim Programmieren aus irgendeinem Grund angenommen werden muß, daß auch 17-Bit-Ergebnisse auftreten können, dann müssen zusätzliche Befehle zum Test des Übertragsbits C nach Abschluß der Addition angefügt werden.

Die Operanden sind wie in Bild 3-5 gezeigt im Speicher untergebracht.

Beachten Sie, daß wir hier angenommen haben, daß der höherwertige Operandenteil „vor“ dem niederwertigen, d.h. an einer niedrigeren Speicheradresse festgehalten werden soll. Das muß nicht unbedingt so sein. In der Tat werden Adressen durch den 6502 gerade umgekehrt gespeichert. Zuerst kommt der niederwertige Teil, in der nächsten Speicherstelle dann der höherwertige. Um für Daten und Adresse dieselbe Vereinbarung zu haben, ist zu empfehlen, auch bei den Daten die höherwertige Hälfte nach der niederwertigen im Speicher festzuhalten. In Bild 3-6 ist dies verdeutlicht.

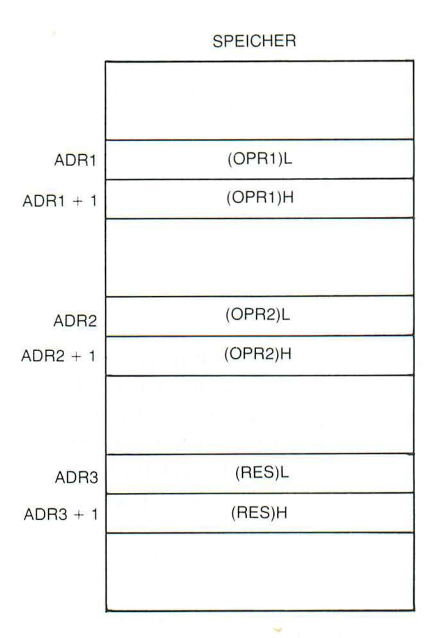


Bild 3-6: In umgekehrter Bytefolge festgehaltene 16-Bit-Operanden

SPEICHER

ADR1-1	(OPR1)L
ADR1	(OPR1)H
ADR2-1	(OPR2)L
ADR2	(OPR2)H
ADR3-1	(RES)L
ADR3	(RES)H

Bild 3-6a: Adressieren der höherwertigen Bytes zuerst

Übung 3.2:
Schreiben Sie das Programm zur 16-Bit-Addition für die Speicherbelegung von Bild 3-6 um.

Übung 3.3:
Nehmen Sie nun an, daß ADR1 nicht wie in Bild 3-6 auf die niederwertige Hälfte von ADR1 sondern auf die höherwertige zeigt, wie es in Bild 3-6a dargestellt ist. Schreiben Sie auch hierzu das passende Additionsprogramm.

Die Entscheidung, wie die 16-Bit-Zahlen gespeichert werden sollen (nieder- oder höherwertiger Teil zuerst) und ob die Adresse sich zuerst auf den niederwertigen oder den höherwertigen Teil bezieht, ist Sache des Programmierers, d.h. Ihre eigene Angelegenheit. Dies ist eine von vielen Möglichkeiten, unter denen Sie beim Entwurf von Algorithmen oder Datenstrukturen abzuwägen lernen müssen.
Wir haben damit gesehen, wie man Dezimalzahlen addiert. Wenden wir uns der Subtraktion zu.

Subtraktion von 16-Bit-Zahlen
Eine 8-Bit-Subtraktion wäre gar zu einfach. Sie können sich daran als Übung versuchen. Wir wollen uns sofort der 16-Bit-Subtraktion zuwenden. Wie üblich seien unsere beiden Zahlen OPR1 und OPR2 unter den Adressen ADR1 und ADR2 festgehalten. Dabei soll der Speicher wie in Bild 3-6 aufgeteilt sein. Zur Subtraktion müssen wir statt des ADC-Befehls den Subtraktionsbefehl SBC verwenden. Ansonsten ist im Vergleich zu den Additionen nur der CLC-Befehl in SEC zu ändern. SEC heißt „set carry to 1", also „Übertragsbit auf 1 setzen". Im 6502 wird dadurch die Tatsache wiedergegeben, daß von der höherwertigen Stelle nicht geborgt worden ist. Der Rest des Programms entspricht genau dem für die Addition. Es hat folgende Form:

```
CLD
SEC             NICHTS GEBORGT
LDA   ADR1      OPR1, NIEDERWERTIGE HÄLFTE
SBC   ADR2      (OPR1 – OPR2), NIEDERWERTIGE HÄLFTE
STA   ADR3      NIEDERWERTIGE ERGEBNISHÄLFTE SPEICHERN
LDA   ADR1+1    OPR1, HÖHERWERTIGE HÄLFTE
SBC   ADR2+1    (OPR1 – OPR2), HÖHERWERTIGE HÄLFTE
STA   ADR3+1    HÖHERWERTIGE ERGEBNISHÄLFTE SPEICHERN
```

Übung 3.4:
Schreiben Sie ein Subtraktionsprogramm für 8-Bit-Zahlen.

Beachten Sie, daß im Fall einer Zweierkomplementrechnung der Endwert der Übertragsflagge C bedeutungslos ist. Wenn der vorgesehene Speicherraum für das Ergebnis nicht ausreicht, wird das Überlaufsbit V im Statusregister P gesetzt. Es kann dann im Bedarfsfall getestet werden.
Wir haben damit einfache binäre Rechnungen ausgeführt. In manchen Anwendungsfällen kann jedoch auch eine andere Zahlendarstellung, die BCD-Form, notwendig werden. In diesem Fall ändern sich die Rechenregeln, was wir uns im folgenden ansehen wollen.

BCD-Artithmetik

8-Bit-BCD-Addition

Die theoretischen Grundlagen der BCD-Arithmetik sind in Kapitel 1 eingeführt worden. Man benutzt sie hauptsächlich im kaufmännischen Bereich, wo jede Stelle unbedingt bis zum Ergebnis erhalten bleiben muß. In BCD-Notation wird jeweils ein Nibble (4 Bits) zur Darstellung einer Dezimalstelle verwendet. Damit lassen sich in einem Byte zwei BCD-Stellen festhalten. (Das nennt man *gepackte BCD-Form.*) Wir wollen nun zwei Bytes mit je zwei BCD-Ziffern addieren.
Um die besondere Problematik herauszufinden, rechnen wir erst einmal ein paar Beispiele durch:
Addieren wir „01“ plus „02“:

„01“ lautet 0000 0001
„02“ lautet 0000 0010
Ergebnis 0000 0011

Das ist die BCD-Darstellung von „03“. (Falls Sie sich bei der Umrechnung unsicher fühlen, können Sie die Tabelle im Anhang zu Hilfe nehmen.) In diesem Fall war alles ganz einfach. Sehen wir uns ein anderes Beispiel an:

„08“ lautet 0000 1000
„03“ lautet 0000 0011

Übung 3.5:
Berechnen Sie die Summe dieser beiden Zahlen in ihrer BCD-Form. Was erhalten Sie? (Antwort folgt.)

Wenn Sie 0000 1011 herausbekommen haben, so haben Sie die *binäre* Summe von „8“ und „3“ berechnet. Sie haben in der Tat „11“ herausbekommen, aber als reine *Dualzahl.* Leider ist „1011“ ein *ungültiger BCD-Kode.* Sie hätten die *BCD-Form* von „11“ erhalten sollen, nämlich 0001 0001!
Das Problem kommt daher, daß die BCD-Darstellung nur die ersten zehn der mit vier Bits möglichen sechzehn Kombinationen zur Wiedergabe von Ziffern „0“ bis „9“ benutzt. Die verbleibenden sechs Möglichkeiten bleiben ungenutzt und sind „verboten“. 1011 ist eine solche verbotene Bitkombination. Mit anderen Worten: Immer wenn die Summe zweier Ziffern größer als „9“ ist, dann muß „6“ zum Ergebnis dazugezählt werden, um die sechs unbenutzten Kodes zu überspringen. Addieren wir als die binäre Darstellung von „6“ zu „1011“:

```
            1011   (verbotenes binäres Ergebnis)
          + 0110   (+ 6)
          ------
Ergebnis: 0001 0001
```

Das ist in der Tat „11“ in BCD-Darstellung! Damit haben wir das richtige Ergebnis erhalten.

Dieses Beispiel verdeutlicht eine der Grundschwierigkeiten bei BCD-Rechnung. Man muß die sechs unbenutzten Kodes kompensieren. Die meisten Mikroprozessoren verlangen für diesen Fall den Einsatz eines besonderen Befehls, „Dezimalkorrektur" (decimal adjust) genannt, mit dem im Falle eines Ergebnisses größer als „9" vom Prozessor eine „6" addiert wird. Beim 6502 geschieht das automatisch durch den ADC-Befehl, vorausgesetzt, man arbeitet im Dezimalmodus. Das ist ein klarer Vorteil dieses Prozessors.

Das nächste Problem läßt sich demselben Beispiel entnehmen. Dort entsteht von der niederwertigen (rechten) BCD-Stelle ein Übertrag in die nächst höherwertige (linke) Stelle. Dieser interne Übertrag muß berücksichtigt und zu der zweiten BCD-Ziffer addiert werden. Der Additionsbefehl des 6502 macht dies automatisch. Jedoch ist es oft ganz nützlich, um diesen internen Übertrag von Bit 3 nach Bit 4 (den „Halbübertrag", half carry) zu wissen. Hierzu gibt es jedoch im 6502 keine Flagge.

Schließlich müssen wie im Fall der binären Addition die Statusbits D und C vor der Rechnung auf den richtigen Wert gesetzt sein. Für das Übertragsbit C ändert sich nichts, jedoch müssen wir jetzt das Dezimalbit D auf 1 setzen, was mit dem Befehl SED (set decimal bit to 1) geschieht. Die Addition der beiden BCD-Zahlen „11" und „22" läßt sich dann so durchführen:

```
CLC              ÜBERTRAG LÖSCHEN
SED              DEZIMALMODUS EINSCHALTEN
LDA    # $11     BCD „11" LADEN
ADC    # $22     BCD „22" DAZU ADDIEREN
STA    ADR       ERGEBNIS UNTER ADR ABLEGEN
```

In diesem Programm benutzen wir zwei neue Symbole: „#" und „$". Mit „#" wird angezeigt, daß als Argument eine *Konstante* und keine Adreßangabe folgt. Das „$"-Zeichen im Operandenfeld gibt an, daß die nachfolgenden Daten als Hexadezimalzahl zu verstehen sind. Die hexadezimale und die BCD-Darstellung der Ziffern „0"

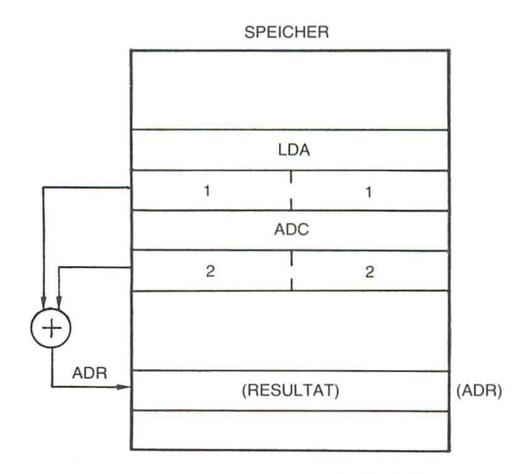


Bild 3-7: Speichern von BCD-Ziffern

bis „9" sind nämlich dieselben. In diesem Fall wollen wir die Konstanten „11" und „22" addieren und das Ergebnis unter der Adresse ADR ablegen. Wenn der Operand als Teil des Befehls wie im Beispiel angegeben wird, so bezeichnet man das als „unmittelbare Adressierung" (immediate addressing). (Die verschiedenen Adressierungsarten werden in Kapitel 5 im Detail behandelt.) Speichern des Ergebnisses unter einer besonders angegebenen Adresse, wie bei STA ADR, nennt man – falls ADR für eine normale 16-Bit-Adresse steht – *absolute Adressierung* (absolute addressing).

Übung 3.6:
Kann man den CLC-Befehl im Programm auch hinter den LDA-Befehl setzen?

BCD-Subtraktion

BCD-Subtraktion ist komplizierter als rein binäre Subtraktion. Man muß hier das *Zehnerkomplement* der Zahl addieren, wie man im binären Fall das *Zweierkomplement* nehmen muß, um eine Subtraktion auszuführen. Das Zehnerkomplement erhält man durch Berechnen des Komplements der Ziffer zu Neun (Neunerkomplement) und anschließendes Addieren von Eins zu der Zahl. In einem Standardmikroprozessor braucht man dazu in der Regel drei bis vier Operationen. Der 6502 verfügt jedoch über einen speziellen Dezimalmodus, in dem eine BCD-Subtraktion mit einem einzigen Befehl durchgeführt werden kann! Zu diesem Zweck wird dem Programm gerade wie im letzten Beispiel ein SED-Befehl (irgendwo) vorangesetzt. Das schaltet den Dezimalmodus ein. Ferner muß man wie bei der binären Subtraktion das Übertragsbit C durch SEC auf Eins setzen. Damit würde ein Programm zur Subtraktion der BCD-Zahlen „26" minus „25" so aussehen:

```
SED             DEZIMALMODUS EINSCHALTEN
SEC             ÜBERTRAG SETZEN (CARRY)
LDA    # $26    BCD „26" IN A LADEN
SBC    # $25    DAVON BCD „25" SUBTRAHIEREN
STA    ADR      ERGEBNIS IN ADR ABLEGEN
```

16-Bit-BCD-Addition

Eine 16-Bit-Addition wird gerade so einfach wie im binären Fall durchgeführt. Das Programm für eine solche Addition folgt:

```
CLC
SED
LDA    ADR1
ADC    ADR2
STA    ADR3
LDA    ADR1-1
ADC    ADR2-1
STA    ADR3-1
```

Übung 3.7:
Vergleichen Sie das obenstehende Programm mit dem zur binären 16-Bit-Addition. Was ist der Unterschied?

Übung 3.8:
Schreiben Sie ein Subtraktionsprogramm für 16-Bit-BCD-Zahlen. (Verwenden Sie weder CLC noch ADC!)

BCD-Flaggen

Im BCD-Modus (Dezimalmodus) gibt die Übertragsflagge bei einer Addition an, daß das Ergebnis größer als „99" ist. Das unterscheidet sich von der Zweierkomplementarithmetik, obwohl die BCD-Ziffern binär dargestellt werden. Bei einer Subtraktion gibt der Wert Null der Übertragsflagge dagegen an, daß nichts von der höherwertigen Stelle geborgt worden ist.

Merke:

- Löschen Sie immer die Übertragsflagge C vor einer Addition!
- Setzen Sie die Übertragsflagge vor einer Subtraktion immer auf 1!
- Wählen Sie den richtigen Rechenmodus: binär oder dezimal!

Befehlsklassen

Wir haben damit drei verschiedene Arten von Mikroprozessorbefehlen verwendet. Da waren LDA und STA, mit denen der Akkumulatorinhalt aus dem Speicher geladen bzw. dort abgelegt worden ist. Diese beiden Befehle gehören zu den *Transferbefehlen.*
Weiter haben wir *arithmetische Befehle* wie ADC und SBC eingesetzt. Mit ihnen haben wir Additions- bzw. Subtraktionsoperationen durchgeführt. Weiter unten werden wir noch mehr Befehle kennenlernen, die die ALU einbeziehen.
Schließlich haben wir solche Befehle wie CLC, SEC u. ä. benutzt, mit denen die Flaggen gehandhabt werden können (in unseren Beispielen das Übertragsbit C und das Dezimalbit D im Statusregister P.) Man nennt sie *Status-* bzw. *Steuerbefehle.* Eine ausführliche Darstellung der 6502-Befehle folgt in Kapitel 4.
Es gibt noch weitere derartige Befehlsklassen in einem Mikroprozessor, die wir bis jetzt noch nicht berücksichtigt haben. Insbesondere sind das die „Branch"- und „Jump"-Sprungbefehle, mit denen die Reihenfolge der Befehlsabarbeitung geändert wird. Diese neuen Befehlsklassen wollen wir im nächsten Beispiel einführen.

Multiplikation

Untersuchen wir ein etwas komplexeres Rechenproblem: Die Multiplikation von Dualzahlen. Um einen Algorithmus hierfür aufzufinden, wollen wir uns zunächst die übliche dezimale Multiplikation ansehen: Multiplizieren wir 12 x 23:

```
Multiplikand  12 x 23  Multiplikator
              -------
                   36  (Teilprodukt)
               +  24
              -------
               =  276  (Endergebnis)
```

Die Multiplikation wird durch Multiplizieren des Multiplikanden mit der niederwertigen Stelle des Multiplikators eingeleitet: „12“ x „3“. Das ergibt das Teilprodukt „36“. Dann multipliziert man die nächst höherwertige Stelle des Multiplikators mit dem Multiplikanden: „12“ x „2“ und addiert das Ergebnis zum vorigen Teilprodukt.
Es muß jedoch noch eine weitere Operation durchgeführt werden: Die „24“ wurde um eine Stelle *nach links geschoben.* Wir hätten genausogut auch sagen können, daß das letzte Teilprodukt („36“) vor der Addition um eine Stelle *nach rechts geschoben* worden ist.
Man erhält also durch Anwenden des kleinen Einmaleins, richtiges Verschieben der Teilprodukte und Addieren das Endergebnis der Multiplikation. Das ist nicht weiter schwierig. Sehen wir uns das mit Dualzahlen an. Eine binäre Multiplikation wird in genau derselben Weise durchgeführt.
Multiplizieren wir z. B. 5 x 3:

```
(MPD) (5) x (3)  (MPR)
      101 x 011
      ---------
          101    (Teilprodukt)
      +  101     (Teilprodukt)
      + 000      (Teilprodukt)
      ---------
      = 01111          (RES)
        (15)
```

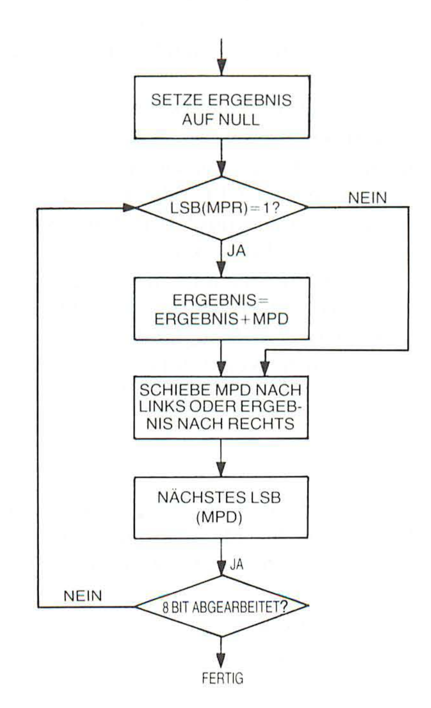


Bild 3-8: Flußdiagramm des Grundalgorithmus zur Multiplikation

Um die Multiplikation auszuführen, gehen wir genau wie im dezimalen Fall vor. Formal läßt sich dieser Algorithmus wie in Bild 3-8 darstellen. Es ist ein Flußdiagramm des Algorithmus, unser erstes Flußdiagramm. Sehen wir es uns genauer an.
Dieses Flußdiagramm ist eine symbolische Wiedergabe des gerade erarbeiteten Algorithmus. Jedes Rechteck steht für eine auszuführende Operation. Es wird in einen oder mehrere Programmbefehle übersetzt werden. Jede Raute gibt einen Test wieder, der an dieser Stelle im Programm auszuführen ist. Dies führt im Programm zu einer Verzögerung. Ist der Test erfolgreich, fahren wir an einer Programmstelle mit der Arbeit fort, fällt er negativ aus, tun wir das an einer anderen Stelle. Die zugehörigen Verzweigungsbefehle werden wir im Programm selbst erläutern. Sie sollten dieses Flußdiagramm jetzt untersuchen und sicherstellen, daß es in der Tat genau den Algorithmus wiedergibt. Beachten Sie den Pfeil, der aus der unteren Raute zur oberen zurückführt. Dies ist notwendig, da wir einen Flußdiagrammteil achtmal durchlaufen müssen, je einmal für jedes Multiplikatorbit. Eine solche Rückkehr zum selben Punkt nennt man aus ersichtlichen Gründen eine *Programmschleife.*

Übung 3.9:
Multiplizieren Sie mit Hilfe dieses Flußdiagrammes die Dualzahlen „4“ und „7“. Stellen Sie sicher, daß Sie wirklich „28“ erhalten haben. Nur wenn Sie das richtige Ergebnis bekommen, sind Sie für die Übersetzung des Diagramms in ein Programm richtig vorbereitet.

Übertragen wir jetzt dieses Flußdiagramm in ein 6502-Programm. In Bild 3-9 ist das vollständige Programm aufgeführt, das wir jetzt im einzelnen untersuchen wollen. Wie wir von Kapitel 1 wissen, bedeutet Programmieren in diesem Fall Übersetzen des Flußdiagramms aus Bild 3-8 in das Programm aus Bild 3-9. Jeder der Kästen des Flußdiagramms ist in einen oder mehrere Befehle zu übertragen.
Es wird angenommen, daß beim Start des Programms Multiplikand MPD und Multiplikator MPR bereits ihre Ausgangswerte besitzen.

```
      LDA   #0        AKKUMULATOR LÖSCHEN
      STA   TMP       LÖSCHEN
      STA   RESAD     LÖSCHEN
      STA   RESAD+1   LÖSCHEN
      LDX   #8        ZÄHLER X
MULT  LSR   MPRAD     MPR LINKS SCHIEBEN
      BCC   NOADD     TESTEN ÜBERTRAGSBIT
      LDA   RESAD     LADE A MIT
                      NIEDERWERTIGEM RES
      CLC             VORBEREITEN ADDITION
      ADC   MPDAD     ADDIERE MPD ZU RES
      STA   RESAD     ZWISCHENSPEICHERN RESULTAT
      LDA   RESAD+1   ADDITION DES RESTANTEILS
                      ZUM GESCHOBENEN MPD
      ADC   TMP
      STA   RESAD+1
NOADD ASL   MPDAD     LINKSSCHIEBEN MPD
      ROL   TMP       MPD-BIT ZWISCHENSPEICHERN
      DEX             ZÄHLER VERMINDERN
      BNE   MULT      WIEDERHOLE FALLS ZÄHLER #0
```

Bild 3-9: 8 x 8-Multiplikation

Der erste Kasten im Flußdiagramm dient zur *Initialisierung*. Wir müssen vor der eigentlichen Arbeit einige Speicherstellen für den weiteren Gebrauch auf „0" setzen. Die vom Multiplikationsprogramm verwendeten Register sind in Bild 3-10 wiedergegeben. Auf der linken Seite der Darstellung steht der für uns hier wichtige Teil des 6502-Prozessors. Auf der rechten Seite befindet sich der in Frage kommende Speicherausschnitt. Wir wollen dabei annehmen, daß die Speicheradressen in der Abbildung von oben nach unten zunehmen. Selbstverständlich könnte man auch die entgegengesetzte Reihenfolge benutzen. Das ganz links stehende X-Register (eines der beiden 6502-Indexregister) wird als *Zähler* benutzt werden. Da wir eine 8-Bit-Multiplikation ausführen, müssen wir 8 Bits des Multiplikanden testen. Leider gibt es beim 6502 keinen Befehl, der es gestattet, diese Bits nacheinander zu untersuchen. Die einzigen bequem testbaren Bits befinden sich im Statusregister P. Wegen dieser bei den meisten Mikroprozessoren vorzufindenden Einschränkungen muß man zur schrittweisen Untersuchung aller Multiplikatorbits seinen Wert in den Akkumulator übertragen. Dessen Inhalt wird dann nach rechts verschoben. Ein Schiebebefehl bewegt alle Bits im Register eine Stelle nach rechts oder nach links. Dabei wird ein Bit aus dem Register herausgeschoben und in das Übertragsbit C übernommen. Diese Arbeitsweise von Schiebebefehlen ist in Bild 3-11 dargestellt. Es sind einige Variationen bezüglich des in das Register hineinzuschiebenden Bits möglich, doch werden wir diese Feinheiten erst in Kapitel 4 bei der Besprechung des 6502-Befehlssatzes untersuchen.

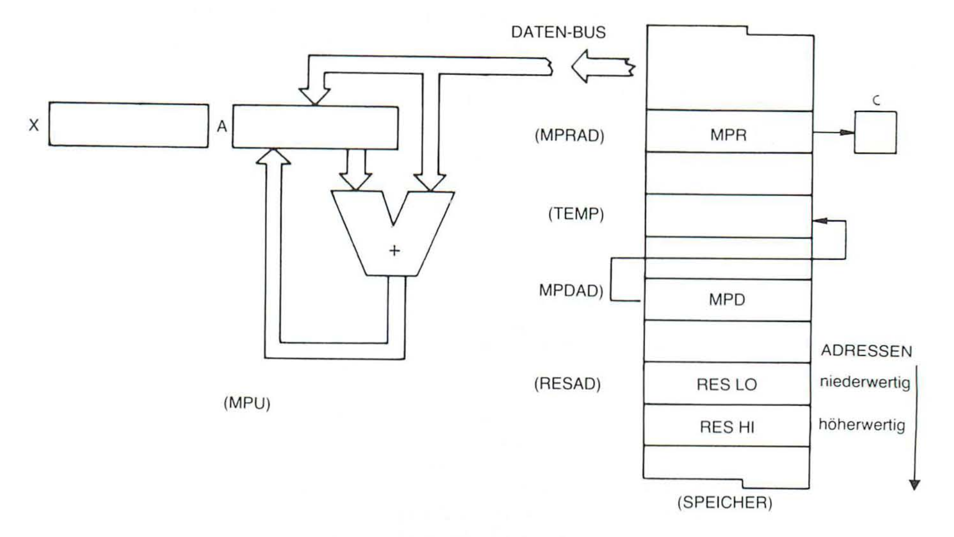


Bild 3-10: Die zur Multiplikation verwendeten Register

Gehen wir zum schrittweisen Test der acht Multiplikatorbits zurück. Da man das Übertragsbit testen kann, wird der Multiplikator achtmal um je eine Stelle verschoben. Dabei gelangt jedesmal das ganz rechts stehende Bit in die Übertragsflagge C, wo sein Wert zum Test bereitsteht.

Das nächste zu lösende Problem ist die Frage des in jedem Schritt aufzuaddierenden Teilprodukts, das insgesamt 16 Bits benötigt, denn eine Multiplikation zweier 8-Bit-Zahlen kann ein 16-Bit-Ergebnis bringen: $2^8 \times 2^8 = 2^{16}$. Daher müssen wir für das Ergebnis 16 Bits reservieren. Leider besitzt der 6502 sehr wenige interne Register, so daß dieses aufaddierte Teilprodukt nicht im 6502 selbst festgehalten werden kann. Ja, wir können wegen der beschränkten Registerzahl weder Multiplikator noch Multiplikand noch das Teilprodukt innerhalb des 6502 unterbringen. Diese Werte müssen alle im Speicher abgelegt werden. Das bewirkt eine langsamere Rechnung, als wenn man alle Operanden und das Teilprodukt im Prozessor selbst festhalten könnte. Es handelt sich dabei um eine 6502-spezifische Einschränkung. Der für die Multiplikation nötige Speicherbereich befindet sich rechts im Bild 3-10. Dort sehen wir oben das für den Multiplikator reservierte Speicherwort. Es möge z.B. eine „3“ enthalten. Diese Speicherstelle ist unter der symbolischen Adresse MPRAD zu erreichen. Darunter finden wir eine „temporäre“ Speicherstelle namens TMP zur Aufnahme von Zwischenergebnissen. Ihre genaue Aufgabe wird unten dargestellt werden. Kurz gesagt, schieben wir hier Stück für Stück den Multiplikanden vor seiner Addition zum letzten Teilprodukt hinein. Der Multiplikand folgt unmittelbar darauf unter der Adresse MPDAD. Er möge den Wert „5“ haben.
Schließlich finden wir am Fuß der Darstellung im Speicher zwei Worte, die Zwischen- bzw. Endergebnis aufnehmen sollen. Ihre Adresse heißt RESAD.

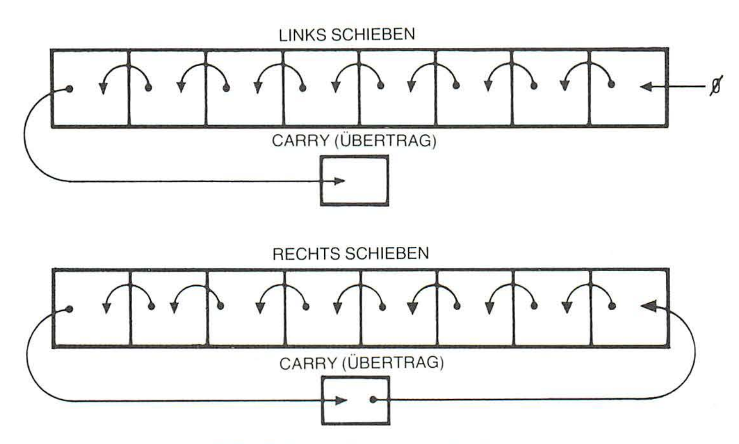


Bild 3-11: Schieben und Rotieren

Diese Speicherstellen sind unsere „Arbeitsregister“, wobei wir in diesem Zusammenhang das Wort „Register“ als Ersatz für „(Speicher)Stelle“ verwenden wollen.
Der am Kopf der Darstellung von MPR nach C reichende Pfeil soll symbolisch angeben, daß der Multiplikator MPR in dieser Richtung in das Übertragsbit C geschoben wird. Natürlich befindet sich dieses C-Bit im 6502-Prozessor und nicht im Speicher.
Gehen wir zum Programm im Bild 3-9 zurück. Die ersten fünf Befehle dort dienen der Initialisation:
Mit den vier ersten Befehlen werden die „Register“ TMP, RESAD und RESAD + 1 gelöscht. Prüfen wir das nach.

```
LDA     0
```

Mit diesem Befehl wird die Konstante „0“ in den Akkumulator geladen. Nach seiner Abarbeitung beträgt der Akkumulatorinhalt binär 00000000.
Dieser Akkumulatorinhalt wird jetzt zum Löschen der drei „Register“ im Speicher benutzt. Erinnern Sie sich, daß das Auslesen eines Werts aus einem Register dessen Inhalt nicht verändert. Wir können ein Register so oft es nötig ist auslesen. Damit können wir weiterarbeiten:

```
STA     TMP
```

Mit diesen Befehlen wird der Akkumulatorinhalt in die Speicherstelle TMP übertragen. Sehen Sie sich zum Verständnis des Datenflusses im Programm Bild 3-10 an. Der Akkumulatorinhalt beträgt gerade 00000000. Das bewirkt, daß Speicherstelle TMP mit lauter Nullen beschrieben wird. Denken Sie weiter immer daran, daß der Akkumulatorinhalt bei dieser Operation seinen Wert „0“ behält. Wir können ihn für die nächsten Schritte weiterbenutzen:

```
STA     RESAD
```

Dieser Befehl arbeitet gerade wie der vorhergehende und löscht den Inhalt unter Adresse RESAD. Machen wir das noch einmal:

```
STA     RESAD + 1
```

Damit haben wir schließlich Speicherstelle RESAD + 1 gelöscht, die den höherwertigen Ergebnisteil aufnehmen soll (Bits 8 bis 15, der niederwertige Teil mit Bits 0 bis 7 kommt in RESAD).

Zum Schluß müssen wir noch eine Bedingung finden, anhand deren wir wissen, daß alle Multiplikatorbits getestet worden sind. Dazu zählen wir die Schiebeoperationen, von denen acht Stück gebraucht werden. Register X wird uns dazu als Zähler dienen und muß auf den Wert „8“ initialisiert werden. Nach jeder Schiebeoperation wird der Zählerinhalt dekrementiert, d.h. um Eins heruntergezählt. Wenn der Zähler den Wert „0“ erreicht hat, sind wir mit der Multiplikation fertig. Setzen wir also dieses Register auf den Wert „8“:

```
LDX     # 8
```

Dieser Befehl lädt Register X mit der Konstanten „8“.
Sehen wir uns das Flußdiagramm in Bild 3-8 an, so finden wir als nächstes die Notwendigkeit, das niederwertige Multiplikatorbit zu testen. Wir haben oben angedeutet, daß das nicht durch einen Befehl allein zu erledigen ist. Wir brauchen zwei Befehle für diesen Zweck. Zunächst muß der Multiplikator eine Stelle nach rechts geschoben und dann das in die Übertragsflagge C herausgeschobene Bit untersucht werden. Führen wir das aus:

```
LSR     MPRAD
```

Dieser Befehl bewirkt ein „logisches Rechtsschieben“ (logical shift right) des Inhalts von Speicherstelle MPRAD.

Übung 3.10:
Nehmen wir an, der Multiplikator habe den Wert „3“. Welchen Wert hat dann das Übertragsbit C nach der Schiebeoperation?

Mit dem nächsten Befehl wird der Wert des Übertragsbits C getestet:

```
BCC     NOADD
```

Das heißt „branch if carry clear“ (Verzweige bei gelöschtem Übertragsbit) zur symbolischen Adresse NOADD.
Hier begegnen wir zum erstenmal einem „Branch“-Befehl, einer Verzweigung im Programm. Bis jetzt wurden alle unsere Programme streng Schritt für Schritt so abgearbeitet, wie wir sie aufgeschrieben hatten. Um logische Tests wie unsere Untersuchung des C-Bits praktisch sinnvoll durchführen zu können, müssen wir die Möglichkeit haben, je nach Testergebnis irgendwo sonst im Programm mit der Befehlsabarbeitung weitermachen zu können. Ein „Branch“-Befehl führt gerade so eine Verzweigungsfunktion aus. Er untersucht den Wert des Übertragsbits C. Ist es gelöscht, d. h. auf „0“ gesetzt, dann verzweigt das Programm zur Adresse NOADD. D.h. der nächste in diesem Fall nach BCC abzuarbeitende Befehl wird von dieser Adresse genommen.
Wenn der Test dagegen negativ ausfällt, d.h. wenn C = „1“ ist, dann erfolgt keine Verzweigung, und die Abarbeitung wird mit dem im Programm unmittelbar auf BCC folgenden Befehl fortgesetzt.
Bezüglich der symbolischen Adresse NOADD ist noch eine Erläuterung angebracht: Es handelt sich dabei um eine „Marke“ im Programm, um ein sogenanntes *Label*. Ein solches steht symbolisch für eine bestimmte Speicheradresse. Zur Entlastung des Programmierers gestattet das Assemblerprogramm die Verwendung derartiger symbolischer Labels anstelle der tatsächlichen Adressen. Während der Übersetzung des Programms, der Assemblierung, ersetzt der Assembler jedes Label durch die zugehörige Speicheradresse. Diese Möglichkeit, symbolische Adressen zu verwenden, verbessert beträchtlich die Lesbarkeit des Programms und setzt den Programmierer in die Lage, zwischen Verzweigungsbefehl und Sprungziel (in unserem Falle: NOADD) beliebig viele Befehle einzufügen, ohne jedesmal alles umschreiben zu müssen. Diese Vorzüge werden bei der Besprechung des Assemblers in Kapitel 10 genauer betrachtet werden.
Um es noch einmal zu sagen: Wenn der Test negativ ausfällt, wird der im Programm unmittelbar folgende Befehl ausgeführt. Sehen wir uns beide Alternativen genauer an:

Alternative 1: C =1

Wenn die Übertragsflagge auf „1“ gesetzt ist, dann fällt der durch BCC ausgeführte Test negativ aus und der nächste Programmbefehl wird abgearbeitet:

LDA RESAD

Alternative 2: C = 0

Der Test ist erfolgreich, und der nächste abzuarbeitende Befehl kommt von der durch das Label NOADD bezeichneten Stelle.
Im Flußdiagramm 3-8 finden wir, daß im Falle C = 1 der Multiplikand zum Teilprodukt (hier den Registern unter RESAD) addiert werden muß. Außerdem ist eine Verschiebung erforderlich. Entweder muß das Teilprodukt eine Stelle nach rechts oder der Multiplikand eine Stelle nach links geschoben werden. Wir wählen hier einen vom handschriftlichen Rechnen gewohnten Weg und schieben den Multiplikanden um eine Stelle nach links. Der Multiplikand befindet sich in den beiden Registern TMP und MPDAD. (Der Einfachheit halber wollen wir hier auch Speicherstellen als „Register" bezeichnen. Dies ist allgemein so üblich.) Die 16 Bits des Teilprodukts stehen unter den Speicheradressen RESAD und RESAD + 1.
Nehmen wir zur Verdeutlichung an, der Multiplikant habe den Wert „5". Die verschiedenen Register finden Sie in Bild 3-10.

Wir müssen einfach nur zwei 16-Bit-Zahlen addieren. Dieses Problem haben wir bereits zu lösen gelernt. (Wenn Sie unsicher sind, sehen Sie sich noch einmal oben den Abschnitt über 16-Bit-Addition an.) Wir werden erst die niederwertigen Bytes und dann die höherwertigen addieren. Fahren wir also fort:

LDA RESAD

Der Akkumulator wird mit dem niederwertigen Teil von RES geladen.

CLC

Vor der Addition müssen wir beim 6502 das Übertragsbit C löschen. Das ist hier unbedingt notwendig, da wir von der Verzweigung her wissen, daß das Übertragsbit auf Eins gesetzt ist. Es ist auf alle Fälle zu löschen.

ADC MPDAD

Der Multiplikand wird zum Akkumulator addiert, der den niederwertigen Teil von RES erhält.

STA RESAD

Das Ergebnis wird in der zugehörigen Speicheradresse als niederwertiger RES-Teil gespeichert. Dann wird die zweite Hälfte der Addition erledigt. Wenn Sie später die Abarbeitung des Programms von Hand überprüfen, sollten Sie nicht vergessen, daß die Addition das Übertragsbit C verändert. Der Übertrag C wird je nach Ergebnis der Addition auf „0" oder „1" gesetzt. Dieser Übertragswert wird im nächsten Additionsschritt automatisch in die Rechnung einbezogen.

Vervollständigen wir zunächst die Addition:

```
LDA     RESAD + 1
ADC     TMP
STA     RESAD + 1
```

Mit diesen drei Befehlen haben wir unsere 16-Bit-Addition abgeschlossen. Der Multiplikand ist zu RES addiert worden. Wir müssen ihn zur Vorbereitung der nächsten Addition jedoch noch eine Stelle nach links verschieben. Allerdings hätten wir den Multiplikanden (außer beim erstenmal) genausogut auch *vor* der Addition verschieben können. Das ist eine der vielen Entscheidungsmöglichkeiten für den Programmierer bei der Programmerstellung.

Schieben wir den Multiplikanden nach links:

```
NOADD ASL     MPDAD
```

Der Befehl gibt ein „arithmetisches Linksschieben" (arithmetic shift left) vor. Er verschiebt den Inhalt von Speicherzelle MPDAD, in der der niederwertige Teil des Multiplikanden steht, um eine Stelle nach links. Das reicht jedoch nicht. Wir dürfen das links aus dem Multiplikandenteil herausgeschobene Bit nicht verlieren. Es steht nach Befehlsausführung im Übertragsbit C. Dort kann es aber nicht bleiben, denn es würde bei der nächsten arithmetischen Operation überschrieben werden. Man muß es in einem anderen Register „dauerhaft" speichern, wozu wir es in TMP schieben wollen. Genau das geschieht beim nächsten Befehl:

```
ROL     TMP
```

Das gibt an: Den Inhalt von TMP nach links „rotieren" (rotate left).
Wir können hier eine interessante Beobachtung machen. Um ein Register einen Schritt nach links zu verschieben, haben wir zwei verschiedene Befehle verwendet: ASL und ROL. Wo liegt da der Unterschied?
Der ASL-Befehl verschiebt den Registerinhalt. Der ROL-Befehl dagegen gibt ein Rotieren an. Er verschiebt den Registerinhalt um eine Stelle nach links und überträgt wie gewohnt das hinausgeschobene Bit in die C-Flagge. Der Unterschied liegt darin, daß hier der *ursprüngliche Wert des Übertragsbits C in die äußerste rechte Bitstelle* eingeschoben wird. In der Mathematik nennt man so etwas eine Ringverschiebung oder Rotation (in unserem Fall eine 9-Bit-Rotation). Das ist genau das, was wir wollen. Durch ROL TMP kommt das zuvor links aus MPDAD herausgeschobene und in C aufbewahrte Bit von dort rechts in TMP hinein. Unser Problem ist damit gelöst.
Der arithmetische Teil des Programms ist damit erledigt. Wir müssen jedoch noch testen, ob wir die Operation achtmal durchgeführt haben, d.h. ob wir mit der Arbeit fertig sind. Wie bei den meisten Mikroprozessoren üblich, brauchen wir dazu zwei Befehle:

```
DEX
```

Mit diesem Befehl wird der Inhalt von Register X dekrementiert. Hatte er zuvor den Wert „8", so sind es nach Ausführen des Befehls „7".

BNE MULT

Das ist noch ein „Branch"-Befehl. Er gibt an, daß zum Label MULT zu verzweigen ist, falls das Ergebnis nicht gleich Null ist (branch if not equal to zero). Solange unser Zählerregister nicht nach Null heruntergezählt ist, verzweigt das Programm automatisch zurück zur symbolischen Adresse MULT. Man bezeichnet das hier als Multiplikationsschleife. Im Flußdiagramm entspricht das dem aus dem letzten Kasten nach oben laufenden Pfeil. Diese Schleife wird achtmal durchlaufen.

MARKE	BEFEHL	X	A	MPR	C	TEMP	MPD	(RESAD)L	(RESAD)H

Bild 3-12: Tabelle für Übung 3.12

Übung 3.11:
Was geschieht, wenn X auf Null heruntergezählt ist? Was für ein Befehl wird als nächster bearbeitet?

In den meisten Fällen wird das eben erstellte Programm ein Unterprogramm sein, dessen letzter Befehl ein Rücksprung RTS zum Hauptprogramm ist. Wir werden den Unterprogrammechanismus später in diesem Kapitel beschreiben.

Ein wichtiger Selbsttest

Wenn Sie das Programmieren erlernen möchten, dann ist es außerordentlich wichtig, daß Sie ein derart typisches Programm bis in alle Einzelheiten verstehen. Der Algorithmus ist vertretbar einfach, aber das Programm ist wesentlich länger als die von uns bis dahin entwickelten Programme. *Es sei Ihnen unbedingt empfohlen, die folgende Übung vollständig und richtig auszuführen, bevor Sie mit dem Kapitel weitermachen.* Wenn Sie sie richtig ausführen konnten, haben Sie den Mechanismus verstanden, mit dem ein Befehl den Inhalt von Speicher- und Prozessorregistern handhabt, und Sie

wissen, wie die Übertragsflagge benutzt wird. Falls Sie nicht damit zu Rande kommen, ist es sehr wahrscheinlich, daß Sie beim Selbstschreiben von Programmen Schwierigkeiten bekommen. Programmieren lernt man nur durch Programmieren. Schieben Sie daher jetzt bitte eine Pause ein, nehmen Sie Papier und Bleistift und arbeiten Sie die folgende Übung durch.

Übung 3.12:
Jedes neu geschriebene Programm sollte erst einmal von Hand auf richtige Ergebnisse überprüft werden. Wir wollen genau das jetzt tun: Gegenstand der Übung ist, die Tabelle aus Bild 3.12 auszufüllen.

Sie können sie unmittelbar beschreiben, besser ist es aber, wenn Sie sich eine Kopie von ihr anfertigen. Die Aufgabe umfaßt das Bestimmen des Inhalts aller wichtigen Register und Speicherstellen nach Abarbeiten jedes Befehls von Anfang bis Ende der Rechnung. Von links nach rechts finden Sie in der Tabelle alle vom Programm belegten Register: X, A, MPR, C (die Übertragsflagge), TMP, MPD, (RESAD) L (der niederwertige RES-Teil) und (RESAD) H (der höherwertige RES-Teil). Im linken Teil müssen das Label, falls vorhanden, und der gerade bearbeitete Befehl (instruction) eingetragen werden. Den rechten Teil sollen Sie mit den Inhalten aller angegebenen Register füllen, wie sie nach Abarbeiten des Befehls vorliegen. Wenn ein Registerinhalt nicht definierbar ist, geben Sie das durch Striche im betreffenden Feld an. Lassen Sie uns die ersten paar Schritte gemeinsam ausfüllen. Die erste Zeile finden Sie in Bild 3-13.

MARKE	BEFEHL	X	A	MPR	C	TEMP	MPD	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----

Bild 3-13: Der erste Befehl des Multiplikationsprogramms

Der erste abzuarbeitende Befehl heißt LDA # 0.
Nach Beendigung des Befehls ist der Inhalt von Register X noch unbekannt. Das wird durch Striche angezeigt. Der Akkumulator ist mit lauter Nullen gefüllt. Wir nehmen weiter an, daß Multiplikator und Multiplikand vor Abarbeiten des Programms geladen worden sind. (Sonst bräuchte man zusätzliche Befehle, um MPR und MPD zu laden.) Wir finden in MPR den binären Wert für „3“ und in MPD den binären Wert für „5“ vor. Das Übertragsbit C ist noch nicht definiert, ebenso Register TMP und die beiden für RESAD benutzten Register. Füllen wir die nächste Zeile aus. Sie finden sie in Bild 3-14: Der einzige Unterschied ist, daß jetzt der Inhalt von Register TMP auf „0“ gesetzt worden ist. Der nächste Befehl setzt (RESAD) L auf „0“ und der übernächste macht dasselbe bei (RESAD) H auf „0“.

MARKE	BEFEHL	X	A	MPR	C	TEMP	MPD	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----
	STA TEMP					00000000			

Bild 3-14: Die beiden ersten Zeilen des Multiplikationsprogramms

Der fünfte Befehl, LDC 8, setzt den Inhalt von X auf „8“. Arbeiten wir noch einen Befehl ab (siehe Bild 3-15).

MARKE	BEFEHL	X	A	MPR	C	TEMP	MPD	(RES)L	(RES)H
	LDA #0	-----	00000000	00000011	--	-----	00000101	-----	-----
	STA TEMP					00000000			
000	STA RESAD							00000000	
	STA RESAD + 1								00000000
	LDX #8	00001000							
MULT	LSR MPRAD			00000001	1				
	BCC NOADD								
	LDA RESAD								
	CLC				0				
	ADC MPDAD		00000101						
101	STA RESAD							00000101	
	LDA RESAD + 1		00000000						
	ADC TEMP								
	STA RESAD + 1								
NOADD	ASL MPDAD						00001010		
	ROL TEMP								
	DEX	00000111							
	BNE MULT								
MULT	LSR MPRAD			00000000	1				
	2. WIEDERHOLUNG								

Bild 3-15: Anfang der Tabelle für Übung 3.12

Der Befehl LSR MPRAD schiebt den Inhalt von MPRAD eine Stelle nach rechts. Sie sehen, daß nach der Verschiebung MPR den Inhalt 0000 0001 hat. Das ganz links stehende Bit von MPR ist in den Übertrag C geschoben worden, der daher jetzt auf „1“ steht. Alle anderen Register bleiben unverändert.

Jetzt sind Sie an der Reihe. Füllen Sie den Rest der Tabelle vollständig aus. Es ist nicht schwer, erfordert aber etwas Aufmerksamkeit. Wenn Sie sich über die Arbeit eines Befehls nicht ganz im Klaren sind, können Sie in der ausführlichen Beschreibung in Kapitel 4 oder im Befehlsverzeichnis im Anhang nachsehen.

Als Endergebnis der Multiplikation sollte binär „15“ in den beiden Registern unter RESAD stehen. Der höherwertige Teil sollte den Wert 00000000, der niederwertige 00001111 enthalten. Wenn Sie auf dieses Ergebnis gekommen sind, haben Sie gewonnen. Wenn nicht, versuchen Sie es noch einmal. Die häufigste Fehlerquelle liegt im falschen Umgang mit dem Übertragsbit. Stellen Sie sicher, daß das C-Bit bei jedem ausgeführten arithmetischen Befehl richtig gesetzt worden ist. Vergessen Sie nicht, daß die ALU das Übertragsbit C nach jeder Addition neu definiert.

Programmalternativen

Das Programm, das wir eben entwickelt haben, ist nur eine der vielen Möglichkeiten, in denen es geschrieben werden kann. Jeder Programmierer findet Mittel und Wege, ein Programm abzuändern und manchmal auch zu verbessern. Zum Beispiel haben wir den Multiplikanden vor der Addition nach links geschoben. Es wäre mathematisch dasselbe, wenn wir das Ergebnis vor Addieren des Multiplikanden um eine Stelle nach rechts schieben würden. Der Vorteil ist, daß wir dann Register TMP nicht

mehr brauchen und so eine Speicherstelle einsparen. Bei einem mit genügend vielen internen Registern ausgestatteten Mikroprozessor wäre das eine vorzuziehende Methode, wenn dadurch MPR, MPD und RESAD im Prozessor selbst stehen könnten. Da wir hier aber ohnehin im Speicher arbeiten müssen, ist die Einsparung einer Speicherstelle nicht von so großer Bedeutung. Die andere Frage lautet aber, ob auf diese Weise die Multiplikation schneller würde. Das ist in der Tat eine interessante Frage.

Übung 3.13:
Schreiben Sie eine 8 x 8-Multiplikation mit dem gleichen Algorithmus, wobei diesmal aber das Ergebnis eine Stelle nach rechts geschoben werden soll und nicht der Multiplikand eine Stelle nach links. Vergleichen Sie das mit dem vorigen Programm und prüfen Sie nach, ob es schneller oder langsamer als dieses abgearbeitet wird.

Es kann hier noch ein Problem auftauchen. Um die Geschwindigkeit des Programms zu ermitteln, bezieht man sich am besten auf die im Anhang bei der Befehlsdarstellung angeführte Anzahl von Taktzyklen, die ein Befehl zur Abarbeitung benötigt. Jedoch hängt bei einigen Befehlen die Zahl der benötigten Taktzyklen von der Lage der Operanden ab. Es gibt für den 6502 mit der „Zero-Page“-Adressierung eine besonders schnell arbeitende Adressierungsart, bei der die Operanden in der ersten Speicherseite (0 bis 255) stehen. Das wird in Kapitel 5 bei den Adressierungsarten erklärt werden. Um es kurz zusammenzufassen: Alle Programme, die schnell abzuarbeiten sind, sollten ihre Variablen in Seite Null stehen haben, da dann der Befehl bei Angabe einer Speicheradresse nur zwei Bytes lang ist (Adressieren von 256 Speicherstellen erfordert gerade ein Byte). Liegen die Operanden sonstwo im Speicher, so werden drei Befehlsbytes benötigt. Die genaue Untersuchung dieser Eigenschaften wollen wir uns jedoch für Kapitel 5 aufheben.

Ein verbessertes Multiplikationsprogramm

Das Programm, das wir vorhin entwickelt haben, ist eine unmittelbare Übersetzung des Algorithmus in den Programmkode. Effektives Programmieren verlangt jedoch einige Aufmerksamkeit für Detailfragen, um so die Programmlänge zu verringern und die Abarbeitungsgeschwindigkeit zu erhöhen. Wir wollen uns jetzt um eine verbesserte Implementation des Algorithmus bemühen.
Eine der befehls- und zeitfressenden Arbeiten ist das Verschieben von Ergebnis und Multiplikator. Ein „Standardtrick“ für einen Multiplikationsalgorithmus beruht auf folgender Beobachtung: Immer wenn man den Multiplikator eine Stelle nach rechts schiebt, wird ganz links ein Bit frei. Gleichzeitig finden wir, daß das erste Zwischenergebnis (oder Teilprodukt) höchstens neun Bits belegt. Nach dem nächsten Multiplikationsdurchlauf wird das Teilprodukt dann wieder um ein Bit verlängert. Mit anderen Worten: Wir können zu Beginn gerade eine Speicherstelle für das Teilprodukt reservieren und dann die vom Multiplikator freigemachten Bitstellen zum Einschieben der nicht mehr zur Rechnung gebrauchten Teilproduktbits verwenden.
Schieben wir also den Multiplikator nach rechts. Das macht links ein Bit frei. In diese Bitstelle übertragen wir das ganz links stehende Bit des letzten Teilprodukts. Sehen wir uns dazu das Programm an.

Wir sollten uns vorher noch über den Einsatz der Prozessorregister Gedanken machen. Die internen Register des 6502-Prozessors stehen in Bild 3-16. Davon wird Register X am besten als Zähler verwendet. Wir benutzen es, um die Anzahl der verschobenen Bits zu zählen. Diese Schiebearbeit kann bei den internen Prozessorregistern (leider) nur der Akkumulator durchführen. Um die Programmeffektivität zu erhöhen, sollten wir hier entweder den Multiplikator oder das Ergebnis festhalten.

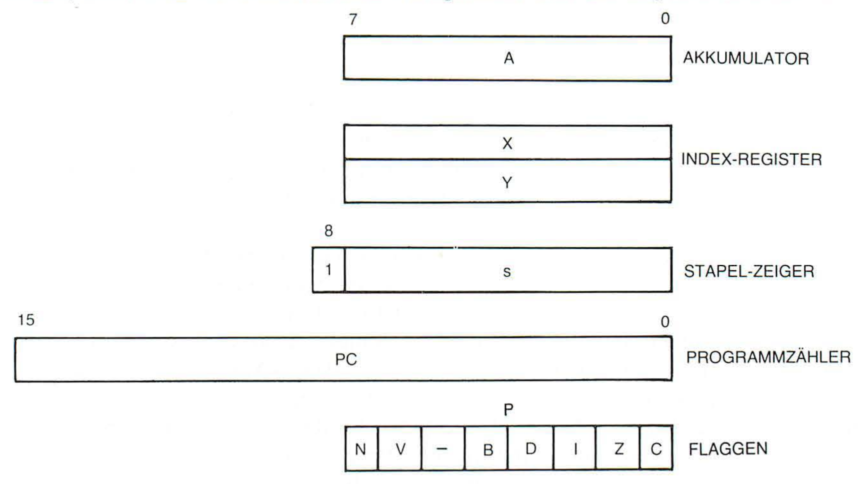


Bild 3-16: Die 6502-Register

Was paßt davon am besten in den Akkumulator? Das Ergebnis muß jedesmal, wenn aus dem Multiplikator eine Eins herausgeschoben wurde, zum Multiplikanden addiert werden. Da der 6502 immer nur etwas zum Akkumulator addiert, legen wir am sinnvollsten das Zwischenergebnis dort hinein.
Die anderen Zahlen müssen im Speicher untergebracht werden (siehe Bild 3-17).

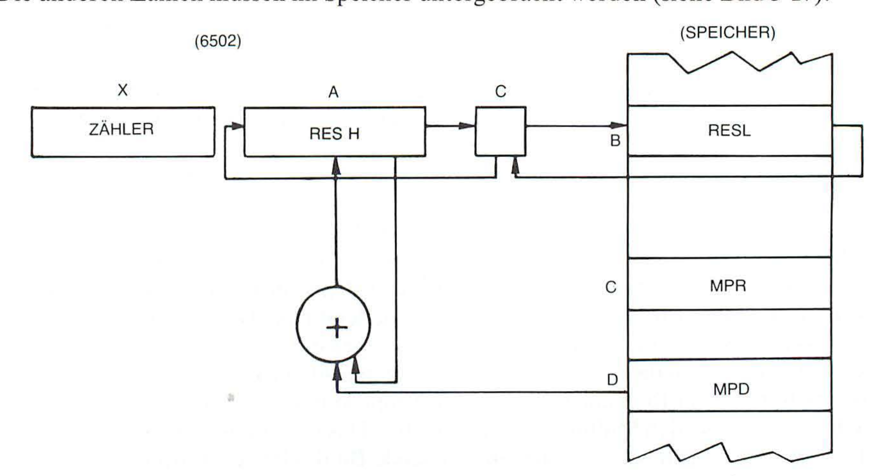


Bild 3-17: Ein verbessertes Multiplikationsprogramm: Die Registerbelegung

A und B enthalten das Ergebnis. In A steht der höherwertige, in B der niederwertige Ergebnisteil. A ist der Akkumulator und B eine Speicherstelle, am besten in Seite Null. C enthält den Multiplikator (eine Speicherstelle). D (auch eine Speicherstelle) trägt den Multiplikanden. Damit kann man das Programm wie in Bild 3-18 abfassen.

```
MULT    LDA  #0      INITIALISIEREN ERGEBNIS MIT NULL
                     (HOEHERWERTIG)
        STA  B       INITIALISIEREN ERGEBNIS
                     (NIEDERWERTIG)
        LDX  #8      X SCHIEBEZAEHLER
WIED    LSR  C       SCHIEBEN MPR
        BCC  NOADD
        CLC          UEBERTRAG = 1. LOESCHEN
        ADC  D       A = A + MPD
NOADD   ROR  A       SCHIEBEN ERGEBNIS
        ROR  B       UEBERTRAGEN BIT NACH B
        DEX          VERMINDERN ZAEHLER
        BNE  WIED    LETZTES SCHIEBEN?
```

Bild 3-18: Ein verbessertes Multiplikationsprogramm

Untersuchen wir das Programm. Da A und B das Ergebnis aufnehmen sollen, müssen sie erst auf Null initialisiert werden. Tun wird das:

```
MULT  LDA  # 0
      STA  B
```

Benutzen wir Register X als Zähler der Bitverschiebungen und initialisieren ihn auf den Wert 8:

```
LDX   # 8
```

Damit können wir wie vorhin in die Multiplikationsschleife (multiplication loop) eintreten. Zunächst verschieben wir den Multiplikator, testen dann das Übertragsbit C, in dem das äußerste rechte Multiplikatorbit Platz gefunden hat. Machen wir das erst einmal:

```
WIED  LSR  C
      BCC  NOADD
```

Hier schieben wir den Multiplikator wie vorher nach rechts. Das entspricht dem Algorithmus, da die Additionsoperationen mathematisch gesehen kommutativ sind. Es gibt jetzt zwei Möglichkeiten: Wurde in Bit C eine Null geschoben, so verzweigen wir zu NOADD. Nehmen wir hier jedoch den anderen Fall an C = 1. Dann können wir so fortfahren:

```
CLC
ADC   D
```

Da der Übertrag den Wert „1“ besitzt, muß er gelöscht werden, bevor wir den Multiplikanden zum Akkumulator addieren. (Dieser enthält die Zwischenergebnisse, im Augenblick noch eine Null.)

Jetzt wäre das Teilprodukt zu verschieben:

```
NOADD ROR    A
      ROR    B
```

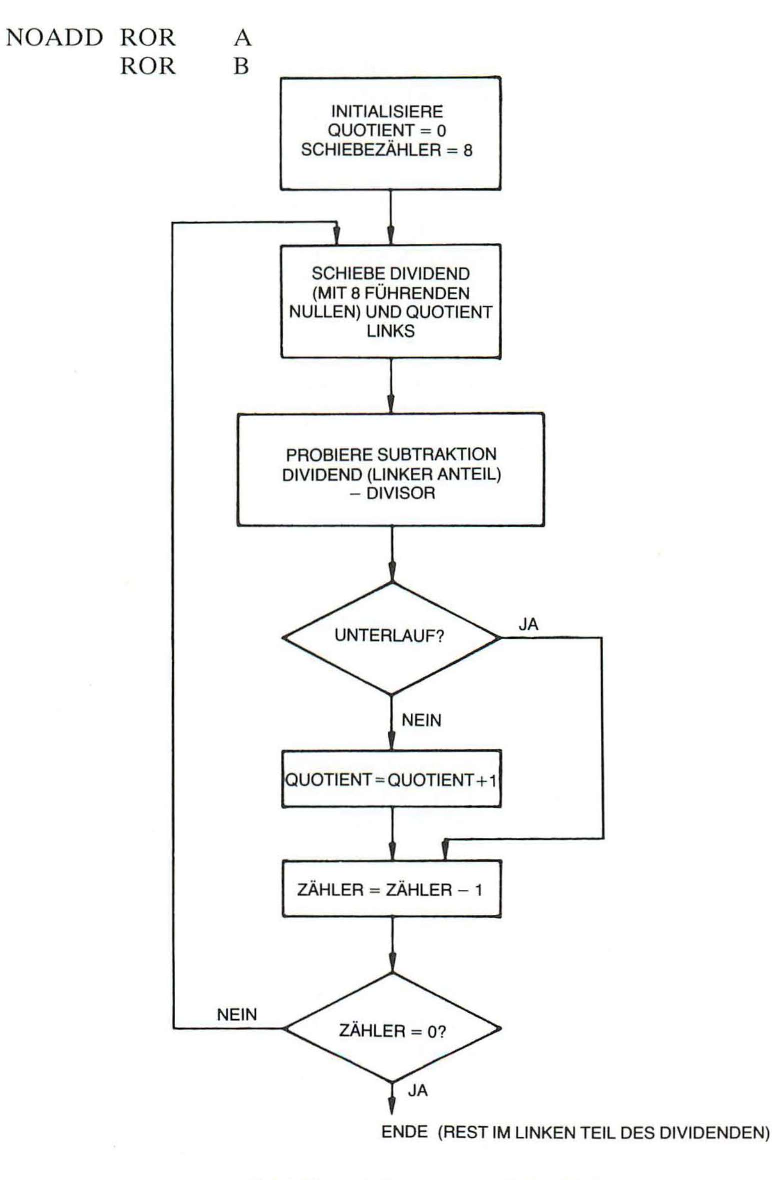


Bild 3-19: Flußdiagramm zur 8-Bit-Division

Das in A stehende Teilprodukt wurde ein Bit nach rechts geschoben. Dabei gelangte das ganz rechts stehende Bit in C. Dieser Übertragsinhalt wurde anschließend in Register B dem niederwertigen Teil des Ergebnisses angefügt.
Wir müssen damit nur noch testen, ob wir fertig sind:

```
DEX
BNE     WIED
```

Wenn wir jetzt unser neues Programm mit dem vorigen vergleichen, so finden wir, daß es gerade halb so viele Befehle umfaßt. Außerdem arbeitet es viel schneller. Damit ist angezeigt, was man durch Auswahl der richtigen Register zur Datenaufnahme erreichen kann.
Ein unmittelbar durchgeführter Entwurf führt zu einem Programm, das arbeitet. Es ergibt jedoch kein optimiertes Programm. Es ist aber ziemlich wichtig, alle verfügbaren Register und Speicherstellen auf die beste Art und Weise zu nutzen. Das Beispiel hat einen brauchbaren Ansatz für eine Registerbelegung mit bestem Ausnutzen der Möglichkeiten verdeutlicht.

Übung 3.14:
Berechnen Sie die Geschwindigkeit einer Multiplikation mit dem zuletzt entwickelten Programm. Nehmen Sie an, daß in fünfzig Prozent aller Fälle eine Verzweigung eintreten wird. Schlagen Sie die für jeden Befehl benötigten Taktzyklen im Anhang nach und legen Sie eine Zyklusdauer von einer Mikrosekunde (pro Taktzyklus) zugrunde.

Binäre Division

Der für die binäre Division benötigte Algorithmus entspricht dem für die Multiplikation. Man subtrahiert den Divisor schrittweise von den jeweils höchstwertigen Dividendenbits. Dabei verwendet man nach jeder Subtraktion das Ergebnis als neuen Dividenden. Der Wert des Quotienten wird gleichzeitig jeweils um Eins erhöht. Unter Umständen kann das Subtraktionsergebnis negativ werden. In diesem Fall muß das vorherige Teilprodukt durch Rückaddieren des Divisors zurückgewonnen und natürlich der Quotient zugleich um Eins vermindert werden. Quotient und Dividend werden dann eine Stelle nach links geschoben und der Algorithmus wiederholt.
Man bezeichnet die eben beschriebene Technik wegen der Rückaddition im Englischen als „restoring method“, als Methode mit Wiedergewinnen des negativ gewordenen Divisors. Es gibt jedoch auch eine Methode, die ohne Rückaddition auskommt und entsprechend „non-restoring method“ genannt wird.

16-Bit-Division

Sehen wir uns eine solche „non-restoring“-Division für einen 16-Bit-Dividenden und einen 8-Bit-Divisor einmal an. Das Ergebnis wird 8 Bits umfassen. Register- und Speicherbelegung für dieses Programm sind in Bild 3-22 dargestellt. Der Dividend steht mit seinem höherwertigen Teil im Akkumulator, mit seinem niederwertigen in

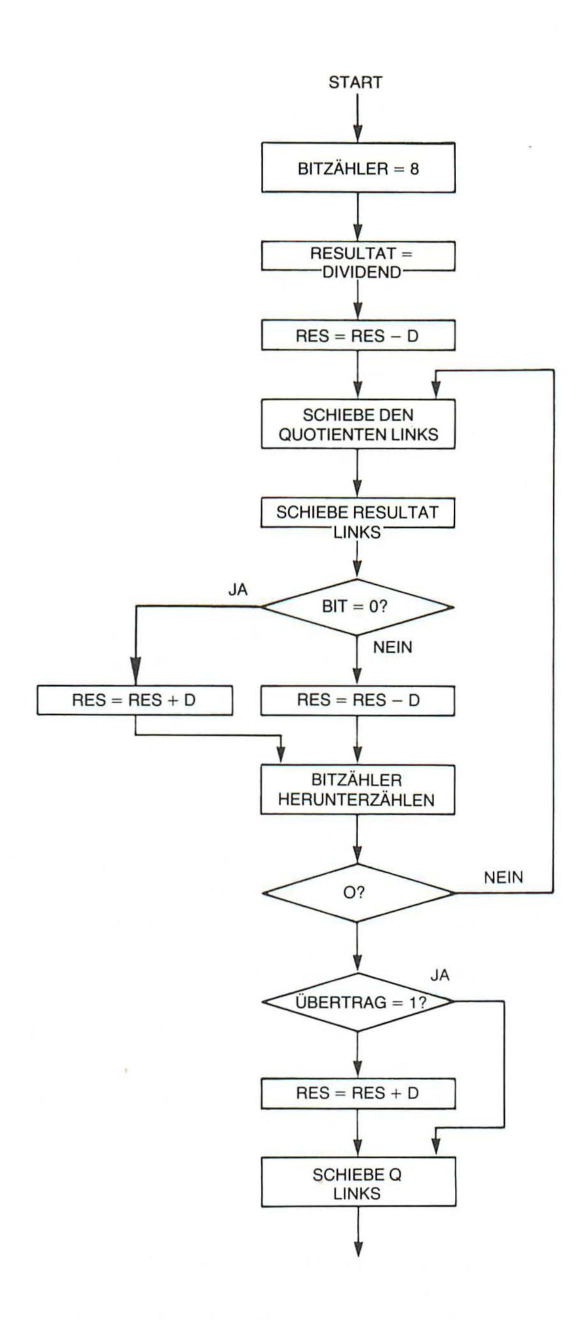


Bild 3-20: Flußdiagramm zur Division einer 16-Bit- durch eine 8-Bit-Zahl

der hier mit 8 bezeichneten Speicherstelle 00. Das Ergebnis wird in Q (Speicherstelle 01) abgelegt. Der Divisor befindet sich in D (Speicherstelle 02). Nach Abschluß der Rechnung steht das Ergebnis in Q und der Rest in A.
Das Programm befindet sich in Bild 3-21, das zugehörige Flußdiagramm in Bild 3-20.

Übung 3.15:
Stellen Sie die richtige Arbeit des Programms durch Abarbeiten von Hand wie in Übung 3.12 sicher. Teilen Sie zu diesem Zweck 33 durch 3. Als Ergebnis sollten Sie 11 in „Register“ Q und einen Rest 0 im Akkumulator erhalten.

Logische Operationen

Außer arithmetischen Operationen kann die ALU im Mikroprozessor noch eine weitere Befehlsklasse, die logischen Verknüpfungen, abarbeiten. Diese umfassen die logischen Verknüpfungen: AND (UND), OR (ODER) und EOR (EXKLUSIV-ODER). Man kann hier auch die bereits erwähnten Schiebebefehle und den beim 6502 CMP (compare) genannten Vergleichsbefehl einordnen. Der Einsatz von AND, OR und EOR wird in Kapitel 4 bei der Besprechung des 6502-Befehlssatzes dargestellt. Stellen wir zur Illustration des Vergleichsbefehls ein kurzes Programm zusammen, das untersucht, ob in Speicherzelle LOC der Wert „0“, „1“ oder sonst etwas steht:

```
        LDA    LOC      INHALT VON LOC LESEN
        CMP    # $00    MIT 0 VERGLEICHEN
        BEQ    NULL     IST 0: VERZWEIGEN
        CMP    # $01    SONST MIT 1 VERGLEICHEN
        BEQ    EINS     IST 1: VERZWEIGEN
KEINES  ...........     KEINES VON BEIDEN
        ...........
NULL    ...........     IST NULL
        ...........
EINS    ...........     IST EINS
        ...........
```

Der erste Befehl LDA LOC liest den Inhalt von Speicherstelle LOC in den Akkumulator. Dieses Zeichen wollen wir testen.

```
LOC     CODE       LINE
0000                          * = $0
0000               B          * = * + 1
0001               Q          * = * + 1
0002               D          * = * + 1
0003                          * = $200
0200    A0 08      DIV        LDY #8
0202    38                    SEC
0203    E5 02                 SBC D
0205    08         WIED       PHP
0206    26 01                 ROL Q
0208    06 00                 ASL B
020A    2A                    ROL A
020B    28                    PLP
020C    90 05                 BCC ADD
020E    E5 02                 SBC D
0210    4C 15 02              JMP NAECH
0213    6502       ADD        ADC D
0215    88         NAECH      DEY
0216    DO ED                 BNE WIED
0218    BO 03                 BCS LETZT
021A    65 02                 ADC D
021C    18                    CLC
021D    26 01      LETZT      ROL Q
021F    00                    BRK
0026                          END
```

Bild 3-21: Programm: Division einer 16-Bit- durch eine 8-Bit-Zahl

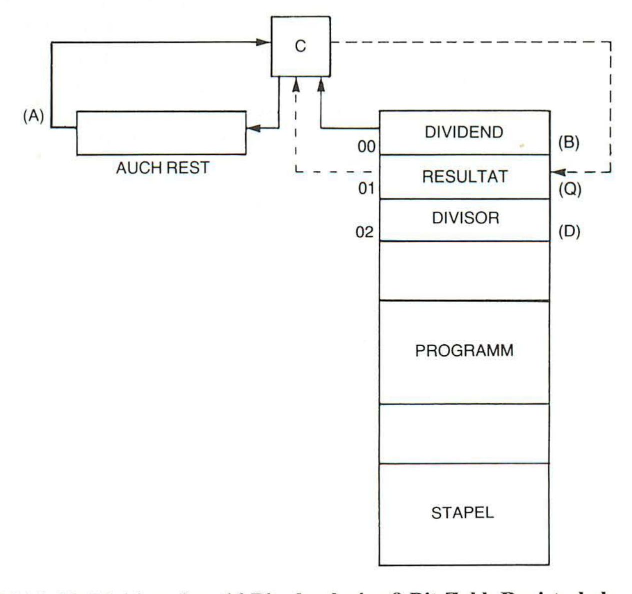


Bild 3-22: Division einer 16-Bit- durch eine 8-Bit-Zahl: Registerbelegung

```
CMP        # $ 00
```

Damit wird der Akkumulatorinhalt mit der hexadezimalen Konstanten „00" d.h. dem Bitmuster 00000000 verglichen. Der Vergleichsbefehl setzt die Z-Flagge im Statusregister P auf Eins, wenn beide Werte gleich sind. Das wird mit dem nächsten Befehl getestet:

```
BEQ        NULL
```

BEQ steht für „branch equal", d.h., „Verzweige bei Gleichheit". Der Befehl untersucht den Wert des Z-Bits und entscheidet so über das Vergleichsergebnis. Ist Z gesetzt, so verzweigt sich das Programm nach NULL. Fällt der Test negativ aus, wird der nächstfolgende Befehl bearbeitet:

```
CMP        # $ 01
```

Der Prozeß wiederholt sich mit einer neuen Vergleichsbasis. Auch hier verzweigt sich das Programm (nach EINS), wenn der Test positiv ausfiel, ansonsten wird der nächstfolgende Befehl abgearbeitet.

Übung 3.16:
Schreiben Sie ein Programm, das den Inhalt von Speicherstelle „24" übernimmt und zur Adresse „STERN" verzweigt, wenn in „24" der ASCII-Kode für „★" (00101010) steht.

Zusammenfassung

Wir haben damit die wichtigsten 6502-Befehle anhand ihrer Verwendung kennengelernt. Wir haben Werte zwischen Speicher und Registern übertragen. Wir haben arithmetische und logische Operationen mit diesen Daten ausgeführt. Wir haben sie getestet und je nach Ausgang des Tests verschiedene Programmabschnitte bearbeitet. Wir haben außerdem im Multiplikationsprogramm eine „Schleife" genannte Programmstruktur eingeführt. Jetzt wollen wir uns eine weitere wichtige Programmstruktur, das Unterprogramm, ansehen.

Unterprogramme

Im Grunde ist ein Unterprogramm nichts weiter als eine selbständige Gruppe von Befehlen, der im Programm ein eigener Name gegeben worden ist. Vom praktischen Standpunkt aus muß ein Unterprogramm mit einem speziellen Befehl eingeleitet werden, der es als solches dem Assembler erkennbar macht. Weiter muß man es durch einen besonderen Rücksprungbefehl namens „*RETURN*" abschließen. Betrachten wir zunächst den Einsatz von Unterprogrammen, um ihren Wert kennenzulernen. Darauf wollen wir untersuchen, wie sie implementiert werden.

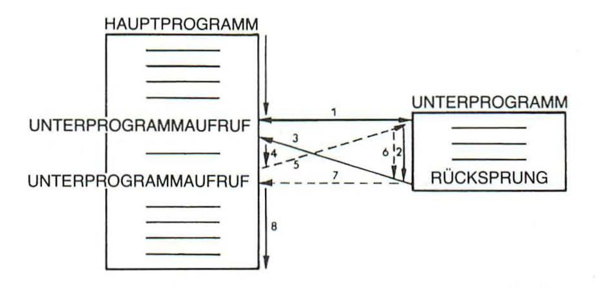


Bild 3-23: Aufruf von Unterprogrammen (Subroutine Calls)

Die Anwendung eines Unterprogramms ist in Bild 3-23 verdeutlicht. Links in der Abbildung ist symbolisch das Hauptprogramm, rechts davon das Unterprogramm dargestellt. Die Befehle des Hauptprogramms werden nacheinander abgearbeitet, bis ein neuer Befehl, CALL SUB, entdeckt wird. Dieser besondere Befehl ist ein *Unterprogrammaufruf* und bewirkt einen Sprung aus dem Haupt- in das angegebene Unterprogramm. Das bedeutet, daß der nächste nach CALL SUB abzuarbeitende Befehl der erste innerhalb des Unterprogramms ist. Dies wird in der Darstellung durch Pfeil 1 wiedergegeben.

Darauf werden die Befehle des Unterprogramms gerade wie die jedes anderen Programms auch abgearbeitet. Wir nehmen dabei an, daß das Unterprogramm selbst keine weiteren Unterprogrammaufrufe enthält. Der letzte Befehl dieses Unterprogramms ist ein RETURN. Dieser Steuerbefehl bewirkt eine Rückkehr der Abarbeitung ins Hauptprogramm. Der nächste nach RETURN abgearbeitete Befehl ist der, der im Hauptprogramm auf CALL SUB folgt. Pfeil 3 verdeutlicht das. Die Programmabarbeitung wird dann wie durch Pfeil 4 angegeben fortgesetzt.

Im Hauptprogramm taucht noch ein zweiter CALL-SUB-Befehl auf, was eine erneute Umschaltung des Programmflusses gemäß Pfeil 5 bewirkt. Das bedeutet, daß das Unterprogramm nach dem CALL-SUB-Befehl erneut abgearbeitet wird.

Immer wenn im Unterprogramm ein RETURN-Befehl entdeckt wird, findet ein Rücksprung zu dem ersten Befehl nach dem zugehörigen CALL SUB statt. Das wird hier durch Pfeil 7 dargestellt. Nach Rückkehr der Abarbeitung ins Hauptprogramm wird die Arbeit normal gemäß Pfeil 8 fortgesetzt.

Damit dürfte die Aufgabe der beiden Steuerbefehle CALL SUB und RETURN klar sein. Doch worin liegt der Wert eines solchen Unterprogramms?

Die wesentlichste Eigenschaft eines Unterprogramms ist die Tatsache, daß es von beliebig vielen Stellen im Hauptprogramm aufgerufen und abgearbeitet werden kann, ohne es jedesmal neu schreiben zu lassen. Als erster Vorteil ergibt sich daraus eine Einsparung von Speicherplatz und die Tatsache, daß man ein Unterprogramm nicht für jeden Einsatz neu abfassen muß. Der zweite Vorteil liegt darin, daß ein Unterprogramm, einmal geschrieben, in den verschiedensten Programmen verwendbar ist. Das spart beim Programmentwurf viel Arbeit ein.

Übung 3.17:
Was ist der Hauptnachteil eines Unterprogramms?

Der Nachteil von Unterprogrammen wird alleine aus der Untersuchung des Programmaufbaus deutlich. Ein mit Unterprogrammen gelöstes Problem wird langsamer abgearbeitet als ein solches ohne Unterprogramm, da bei jeder Verwendung zwei Befehle zusätzlich bearbeitet werden müssen: der Unterprogrammaufruf CALL SUB und der Rücksprungsbefehl RETURN.

Implementation des Unterprogrammechanismus

Untersuchen wir, wie die beiden Steuerbefehle CALL SUB und RETURN innerhalb des Prozessors implementiert sind. Der Aufruf CALL SUB bewirkt, daß der nächste Befehl von einer neuen Adresse geholt wird. Erinnern Sie sich (oder lesen Sie noch einmal Kapitel 2), daß die Adresse des nächsten Befehls im Programmzähler PC steht. Das heißt, daß für die Abarbeitung von CALL SUB der Inhalt des Programmzählers verändert werden muß. Der Befehl bewirkt das Laden einer neuen Adresse in den Programmzähler. *Ist das aber wirklich genug?*
Die Antwort auf diese Frage ergibt sich, wenn wir uns den anderen zu implementierenden Befehl ansehen: den Rücksprungbefehl RETURN. Der RETURN-Befehl bewirkt, wie sein Name sagt, die Rückkehr der Programmabarbeitung zu dem auf CALL SUB folgenden Befehl. Das geht aber nur, wenn die Adresse dieses Befehls vorher irgendwo festgehalten worden war. Diese Adresse ist gerade der Programmzählerwert nach Übernahme des CALL-SUB-Befehls. Denn der Programmzähler wird (siehe Kapitel 2) nach jedem Einsatz automatisch um Eins weitergezählt. Und genau diese Adresse müssen wir irgendwo aufheben, um sie später für den RETURN-Befehl zur Verfügung zu haben.
Das nächste Problem ist: Wo können wir die Rückadresse speichern? Sie muß an einer Stelle abgelegt werden, an der sie mit Sicherheit nicht gelöscht oder sonstwie verändert werden kann. Betrachten wir dazu die in Bild 3-24 wiedergegebene Situation. Hier ruft Unterprorgamm SUB1 seinerseits ein Unterprogramm, SUB2, auf.

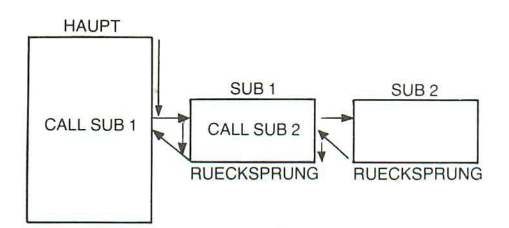


Bild 3-24: Verschachelte Unterprogramme

Unser Mechanismus muß auch diesen Fall verarbeiten können. Natürlich können noch mehr derartige ineinanderverschachtelte Aufrufe, sagen wir N Stück, auftreten. Bei jedem im Programm neu entdeckten CALL SUB muß der Mechanismus den Programmzählerwert aufs neue speichern. Das beinhaltet, daß wir für diesen Speicher bei N ineinander verschachtelten Unterprogrammen mindestens 2N Speicherstellen für die Rücksprungadressen brauchen. Außerdem muß das in geordneter Reihenfolge geschehen. In unserem Beispiel müssen wir zuerst aus SUB2 und dann aus SUB1

zum aufrufenden Programm zurückkehren. Mit anderen Worten brauchen wir eine Struktur, die die chronologische Folge der anfallenden Daten bewahrt.

Diese Struktur hat einen Namen. Wir sind ihr bereits begegnet. Es ist der *Stapel.* Bild 3-26 zeigt den Inhalt eines solchen Stapels während aufeinanderfolgender Unterprogrammaufrufe. Betrachten wir zunächst das Hauptprogramm. Auf Adresse „100" wird der erste Unterprogrammaufruf entdeckt: CALL SUB1. Wir nehmen an, daß beim betrachteten Mikroprozessor dieser Unterprogrammaufruf drei Bytes erfordert. Der nächste Befehl im Hauptprogramm steht dann nicht unter Adresse „101", sondern unter „103". Weil die 6502-Steuereinheit „weiß", daß es sich um einen Dreibytebefehl handelt, hat der Programmzähler nach vollständiger Befehlsübernahme den Wert „103". Im Ergebnis der Aufrufsbearbeitung wird die Startadresse von SUB1, „280", in den Programmzähler geladen.

Die zweite Arbeit des Unterprogrammaufrufs besteht darin, den alten Wert „103" des Programmzählers auf dem Stapel abzulegen. Das ist in Bild 3-26 dargestellt, wo zum Zeitpunkt 1 auf dem Stapel der Wert „103" gespeichert ist. Gehen wir in der Abbildung einen Schritt nach rechts. Auf Adresse „300" wird ein neuer Unterprogrammaufruf entdeckt. Gerade wie vorher wird dadurch „900", die Anfangsadresse von SUB2, in den Programmzähler geladen. Gleichzeitig wird der Programmzählerstand „303" auf den Stapel gebracht. Aus Bild 3-26 ist zu entnehmen, daß zum Zeitpunkt 2 als weiterer Wert „303" im Stapel steht. Die Abarbeitung wird dann rechts in Bild 3-25 mit SUB2 fortgesetzt.

Damit können wir die Wirkungsweise des Rücksprung-Befehls (RETURN) untersuchen und die richtige Arbeit des Stapelmechanismus nachprüfen. Die Abarbeitung von SUB2 verläuft ohne weitere Störungen bis zum Zeitpunkt 3 ein Rücksprung-Befehl (RETURN) entdeckt wird. Dadurch wird einfach nur das Spitzenelement vom Stapel heruntergenommen und in den Programmzähler geladen. Mit anderen Worten: Der Programmzähler erhält seinen Wert vor Beginn der Arbeit an SUB2 zurück. In unserem Beispiel steht auf der Stapelspitze die Adresse „303".

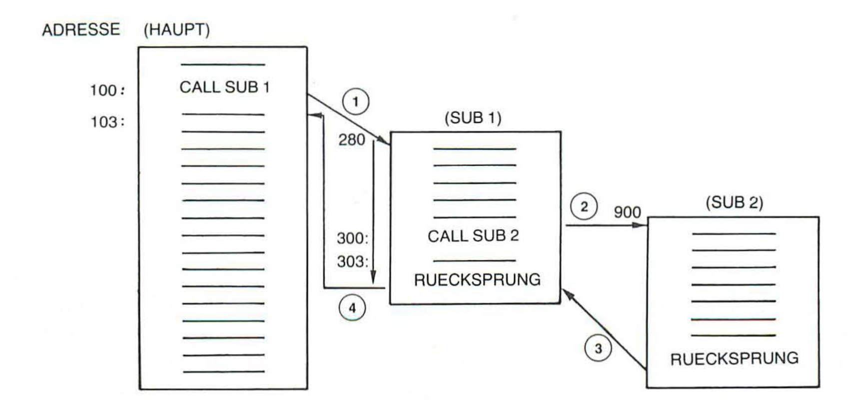


Bild 3-25: Abarbeitung von Unterprogrammen

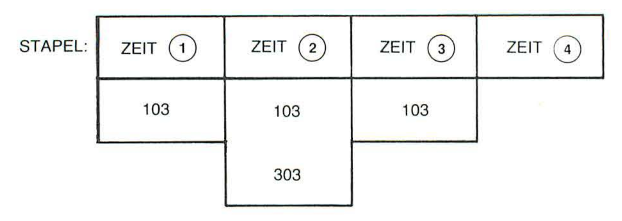


Bild 3-26: Stapelinhalt während der Unterprogrammabarbeitung

Bild 3-26 zeigt, daß zum Zeitpunkt 3 dieser Wert aus dem Stapel entfernt worden ist. Daraufhin wird die Arbeit mit dem Befehl unter Adresse 303 fortgesetzt und geht in SUB1 weiter, bis zum Zeitpunkt T4 der RETURN-Anschluß von SUB1 vorgefunden wird. Das bewirkt das Übertragen des Inhalts „103" der Stapelspitze in den Programmzähler und damit die Fortsetzung der Arbeit im Hauptprogramm ab dieser Stelle. Genau diese Ablauffolge wollten wir auch haben. Laut Bild 3-26 ist der Stapel ab dem Zeitpunkt T4 leer. Der Mechanismus arbeitet korrekt.

Dieser Unterprogrammechanismus arbeitet, bis der Stapel voll ist. Aus diesem Grund waren die früheren Mikroprozessoren mit einem internen Hardwarestapel aus vier oder acht Registern im wesentlichen auf eine Schachtelungstiefe von vier oder acht Unterprogrammen beschränkt. Theoretisch ist der 6502 mit seinem auf 256 Bytes beschränkten Stapelbereich auf Seite 1 zur Verarbeitung von maximal 128 verschachtelten Unterprogrammen in der Lage. Das gilt aber nur, wenn keine Programmunterbrechungen (interrupts) vorkommen und der Stapel nicht als Variablenspeicher benutzt wird. In der Praxis erreicht man diese Schachtelungstiefe jedoch nur selten.

Beachten Sie, daß in Bild 3-24 und 3-25 die Unterprogramme rechts vom Hauptprogramm dargestellt worden sind. Das geschah nur der Übersicht halber. In Wirklichkeit werden die Unterprogramme im Verlauf der Programmierarbeit wie andere Programmteile auch angegeben. Auf dem Papier können die Unterprogramme am Anfang, in der Mitte oder am Ende des Hauptprogramms stehen. Aus diesem Grund müssen sie für den Assembler mit Hilfe eines besonderen Befehls gekennzeichnet sein. Derartige *Assembleranweisungen* (directives) werden in Kapitel 9 vorgestellt.

Unterprogramme beim 6502

Wir haben damit den Unterprogrammechanismus kennengelernt und wissen, wie man den Stapel zu seiner Implementation verwendet. Der 6502-Befehl zum Unterprogrammaufruf heißt JSR (jump to subroutine – zum Unterprogramm springen) und umfaßt drei Bytes. Leider handelt es sich dabei um einen *unbedingten Sprung:* Es werden zu seiner Ausführung keinerlei Bedingungen getestet. Will man ihn von bestimmten Flaggen abhängig machen, so muß man ihn einen „Branch"-Befehl voransetzen.

Der Rücksprung aus dem Unterprogramm wird beim 6502 durch RTS (return from subroutine – aus dem Unterprogramm zurückkehren) befohlen. Es handelt sich dabei um einen Einbytebefehl.

Übung 3.18:
Warum dauert die Abarbeitung des Rücksprungbefehls genauso lange wie der Aufruf des Unterprogramms, obwohl es sich um einen Einbytebefehl handelt? (Hinweis: Wenn die Antwort nicht klar ist, sehen Sie sich die Implementation des Unterprogrammechanismus noch einmal an und analysieren Sie die dabei durchgeführten internen Arbeiten.)

Unterprogrammbeispiele

Die meisten der von uns entwickelten Programme und diejenigen, die wir noch erstellen werden, dienen normalerweise als Unterprogramme. Zum Beispiel dürfte das Multiplikationsprogramm sicher in den verschiedensten Teilen eines großen Programms anzuwenden sein. Das wird ermöglicht, wenn man es als Unterprogramm mit (beispielsweise) dem Namen MULT definiert und bei Bedarf mittels JSR MULT aufruft. Am Programmende von MULT ist dazu noch der Rücksprungbefehl JSR anzufügen.

Übung 3.19:
Verändert der Einsatz von MULT als Unterprogramm im Hauptprogramm irgendwelche Register oder Flaggen?

Rekursive Unterprogrammaufrufe

Rekursion bedeutet, daß ein Unterprogramm sich selbst aufruft. Wenn Sie den Implementationsmechanismus für Unterprogramme verstanden haben, dann sollten Sie die folgende Frage beantworten können:

Übung 3.20:
Ist es zulässig, wenn ein Unterprogramm sich selbst aufruft? (Mit anderen Worten: Arbeitet alles auch dann noch richtig, wenn das Unterprogramm sich selbst aufruft?) Wenn Sie sich darüber nicht klar sind, zeichnen Sie den Stapel auf und füllen Sie ihn mit den anfallenden Rücksprungadressen. Sie werden dann unmittelbar feststellen können, in welchem Fall es funktioniert. Das beantwortet die Frage. Sehen Sie sich auch die Register und den Speicher auf mögliche Probleme an (vgl. Übung 3-19).

Unterprogrammparameter

Ein Unterprogramm soll in der Regel irgendwelche Daten aus dem und für das aufzurufende Programm bearbeiten. Bei der Multiplikation z.B. müssen dem Unterprogramm die beiden miteinander malzunehmenden Zahlen übergeben und von diesem das Produkt zurückgeliefert werden. Wir hatten bei der Besprechung des Programms festgelegt, daß beim Unterprogrammaufruf Multiplikand und Multiplikator in bestimmten Speicherstellen liegen müssen. Das verdeutlicht eine der Methoden, derartige *Parameter* zu übergeben: durch Speicherstellen. Es sind noch zwei weitere Techniken üblich, so daß Parameter auf drei Arten übergeben werden können:

1. durch Register, 2. durch den Speicher, 3. durch den Stapel.

– *Register* lassen sich zur Parameterübergabe verwenden. Diese Lösung hat viele Vorzüge, vorausgesetzt, daß eine genügende Zahl an Prozessorregistern vorhanden ist. In diesem Fall braucht man keine feste Speicherstelle vorzusehen. Das Unterprogramm bleibt unabhängig vom Speicher. Würde man eine feste Speicherstelle benutzen, so hätte der Benutzer des Unterprogramms sehr sorgfältig darauf zu achten, daß er dieselbe Vereinbarung benutzt und daß der Speicher auch wirklich verfügbar ist (sehen Sie sich noch einmal Übung 3-20 an). Das ist der Grund, warum in vielen Fällen ein ganzer Speichenblock von vornherein nur zur Parameterübergabe reserviert wird.
– *Einsatz des Speichers* hat den Vorteil größerer Flexibilität (mehr Daten), ist aber nicht so leistungsfähig wie der erste Fall und bindet das Unterprogramm an einen bestimmten Speicherbereich.
– *Die Parameterübergabe durch den Stapel* hat denselben Vorteil wie die Übergabe durch Register. Sie ist speicherunabhängig. Das Unterprogramm braucht nur zu wissen, daß es beispielsweise zwei Werte über den Stapel empfangen wird. Das hat natürlich den Nachteil, daß der Stapel mit Daten vollgestopft und so die Zahl der zu verschachtelnden Unterprogramme beeinträchtigt wird.
Die Entscheidung liegt beim Programmierer. Üblicherweise zieht man es vor, speicherunabhängig zu arbeiten, so lang es nur geht.

Wenn keine Register zur Parameterübergabe bereitstehen, ist die nächstbeste Lösung in aller Regel der Stapel. Wenn dabei jedoch sehr viele Parameter an ein Unterprogramm übergeben werden sollen, dann muß die Information im Hauptspeicher stehen. Eine elegante Möglichkeit zur Lösung des Problems besteht in der Übergabe eines *Zeigers* auf den Datenblock anstelle der Daten selbst. Ein Zeiger kann in einem Register übergeben (was beim 6502 diesen allerdings auf 8 Bits einschränkt) oder auf dem Stapel abgelegt werden (dann werden zwei Stapelelemente für 16 Bits gebraucht).
Wenn schließlich keine dieser beiden Lösungen anwendbar ist, dann kann man bei Aufstellen des Unterprogramms vereinbaren, daß es den Zeiger an einer bestimmten Speicherstelle (dem „Briefkasten“) abholt.

Übung 3.21:
Welche der drei oben beschriebenen Methoden eignet sich am besten zur Rekursion?

Unterprogrammbibliotheken

Das Aufteilen eines Programms in selbständige Unterprogramme hat einen sehr großen Vorteil: Man kann die Untereinheiten für sich selbst überprüfen und von Fehlern befreien (debugging) und man kann ihnen jeweils einen mnemonischen Namen, eine leicht zu identifizierende Bezeichnung geben, unter der sie aufgerufen werden. Wenn derartige Untereinheiten in mehreren Programmen verwendbar sind, so kann man sich eine ganze Bibliothek allgemein brauchbarer Unterprogramme aufbauen. Es gibt beim Programmieren allerdings keine Allheilmittel. Setzt man Unterprogramme systematisch an jeder nur denkbaren Stelle ein, so erhält man leicht ein Programm mit

ziemlich schlechter Leistungsfähigkeit. Ein guter Programmierer wird immer zwischen den Vor- und Nachteilen des Unterprogrammeinsatzes abzuwägen haben.

Zusammenfassung

In diesem Kapitel wurden die Methoden dargestellt, mit denen man innerhalb des 6502 Information mit Hilfe von Befehlen handhaben kann. Es wurden Algorithmen mit wachsender Komplexität vorgestellt und in Programm übersetzt. Alle wichtigen Befehlsklassen wurden verwendet.
Des weiteren wurden wichtige Programmstrukturen wie Schleifen, Stapel und Unterprogramme definiert.
Sie sollten an dieser Stelle die wichtigsten Grundlagen des Programmierens verstanden haben und die für Standardanwendungen wichtigsten Techniken beherrschen. Als nächstes wollen wir uns die beim 6502 verfügbaren Befehle ansehen.

KAPITEL 4
DER 6502-BEFEHLSSATZ

Teil 1: Allgemeine Beschreibung

Einleitung

In diesem Kapitel werden zunächst die verschiedenen Befehlsklassen beschrieben, die in einem Allzweckcomputer verfügbar sein sollten. Danach werden nacheinander sämtliche beim 6502 vorhandenen Befehle untersucht und dargestellt, welche Operationen sie ausführen, die Art, in der sie die Flaggen beeinflussen, und ihr Einsatz unter den verschiedenen Adressierungsarten. Eine detaillierte Untersuchung der Adressierungsarten findet sich dann in Kapitel 5.

Befehlsklassen

Man kann hier auf vielerlei Art klassifizieren. Es gibt hier keinen allgemeinverbindlichen Standard. Wir werden fünf Befehlsklassen unterscheiden:

1. Transferbefehle,
2. Verarbeitungsbefehle,
3. Test- und Verzweigungsbefehle,
4. Ein/Ausgabebefehle,
5. Steuerbefehle.

Untersuchen wir, was es damit auf sich hat.

Transferbefehle

Transferbefehle übertragen 8-Bit-Daten zwischen zwei Registern, zwischen Register und Speicherstelle oder zwischen Register und Ein/Ausgabeeinheit. Es kann für Register mit besonderen Aufgaben eigene Transferbefehle geben, wie z. B. die Stapeloperationen „push“ und „pull“, die ein Datenwort zwischen Stapelspitze und Akkumulator übertragen und gleichzeitig den Stapelzeiger neu setzen.

Verarbeitungsbefehle

die Befehle zur Datenverarbeitung lassen sich in vier Untergruppen einteilen:

- Arithmetische Operationen (z.B. Addition, Subtraktion),
- Logische Operationen (wie UND, ODER, EXKLUSIV-ODER),
- Schiebeoperationen (Verschieben, Rotieren),
- Inkrementieren und Dekrementieren.

Es wäre für eine leistungsfähige Datenverarbeitung wünschenswert, komplexere arithmetische Operationen wie Multiplikation oder Division zur Verfügung zu haben. Die meisten 8-Bit-Mikroprozessoren bieten diese Möglichkeit leider nicht an (wohl aber die neueren 16-Bit-Prozessoren). Ebenso hätte man gerne leistungsfähigere Schiebeoperationen, wie Befehle zur Verschiebung um n Bits oder Austausch der beiden Nibbles im Byte (Vertauschen der beiden Bytehälften). Leider findet man auch das in den wenigsten 8-Bit-Prozessoren.
Die meisten der Operationen sprechen für sich. Wir wollen uns hier nur noch einmal mit dem Unterschied zwischen einer *Verschiebung* und einer *Rotation* beschäftigen (Bild 4-1). Eine Verschiebeoperation schiebt den Inhalt eines Registers oder einer Speicherstelle um ein Bit nach links oder rechts. Das aus dem Register herausgeschobene Bit kommt in die Übertragsflagge C (carry). Das auf der anderen Seite frei werdende Bit wird auf „0" gesetzt (d.h. es wird dort eine Null „hineingeschoben").

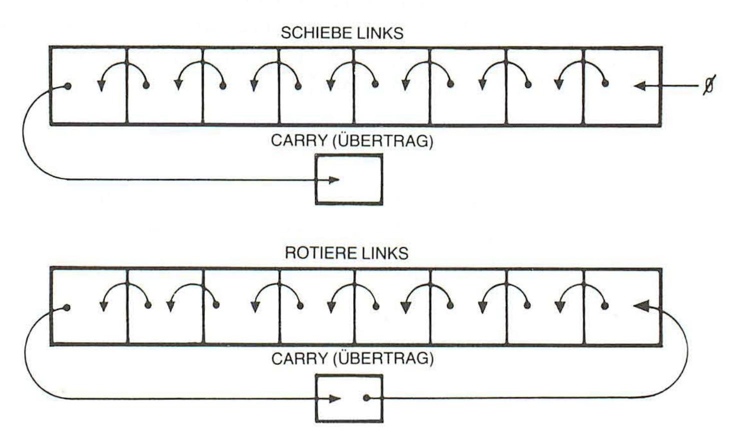


Bild 4-1: Verschieben und Rotieren von 8-Bit-Worten

Im Fall der Rotation kommt das herausgeschobene Bit ebenso in den Übertrag C. Jedoch wird jetzt in die freigewordene Bitstelle der vorige Wert von C geschoben. Das ergibt ein Rotieren des 9-Bit-Inhalts von Register und Übertrag. In vielen Fällen wäre demgegenüber eine 8-Bit-Rotation wünschenswert, bei der das auf der einen Seite des Registers herausgeschobene Bit auf der anderen wieder hineingeschoben wird. Üblicherweise findet man diese Form bei Mikroprozessoren nicht.

Schließlich gibt es beim Rechtsverschieben des Registerinhalts noch die oftmals nützliche Form mit „Vorzeichenkopie", die sogenannte *arithmetische Verschiebung*. Diese erhält ihre Bedeutung beim Verschieben von Zweierkomplementzahlen, insbesondere beim Erstellen von Fließkommaprogrammen, bei denen oft negative Zahlen verarbeitet werden müssen. Wenn man eine Zweierkomplementzahl nach rechts verschiebt, sollte das ganz links stehende Bit mit dem Wert des Vorzeichens aufgefüllt werden, bei negativen als mit einer „1". Leider gibt es diese Schiebemöglichkeit im Gegensatz zu anderen Prozessoren nicht beim 6502.

Tests und Verzweigungen

Testbefehle untersuchen den Stand der Flaggen im Statusregister auf „0", „1" oder bestimmte Kombinationen mehrerer Flaggen. Es ist daher wünschenswert, in diesem Register so viele Flaggen wie nur möglich zu haben. Außerdem wäre es angenehm, könnte man solche Bits mit einem einzigen Befehl testen. Schließlich wäre es wünschenswert, den Wert eines jeden Bits beliebiger Register untersuchen zu können oder den Wert eines Registers mit dem jedes anderen nach Wunsch zu vergleichen (größer als, gleich, kleiner als). Mikroprozessortestbefehle sind in der Regel auf die Untersuchung einzelner Bits im Statusregister eingeschränkt.

Die Sprung- bzw. Verzweigungsbefehle, die vorhanden sein können, gliedern sich in der Regel in drei Kategorien:

- den vollen Sprung, der eine 16-Bit-Adresse benötigt,
- die Verzweigung, die oft auf einen 8-Bit-Abstand als Adreßangabe (displacement) eingeschränkt wird, und
- den Unterprogrammaufruf.

Bequem wären Verzweigungen mit zwei oder drei Wegen, die z.B. davon abhängen können, ob bei einem Vergleich „größer als", „gleich" oder „kleiner als" herausgekommen ist. Außerdem wären kurze Sprünge (skips) vorwärts oder rückwärts über einige wenige Befehle in vielen Fällen sinnvoll. Schließlich erfordern viele Schleifen als Schleifenendbedingung das Herauf- oder Herunterzählen eines Registers mit anschließendem Sprung zum Schleifenanfang, wenn die Schleifenendbedingung nicht erreicht ist. Für eine wirkungsvolle Schleifenprogrammierung wäre ein einfacher Befehl, der das Zählen und den eventuell notwendigen Sprung übernimmt, sehr nützlich. Die meisten Mikroprozessoren bieten diese Möglichkeit jedoch leider nicht an. In aller Regel hat man die Möglichkeit zu einfachen Verzweigungen, kombiniert mit Einflaggentests. Damit wird natürlich das Programmieren verkompliziert und die Leistungsfähigkeit des Programms verringert.

Ein/Ausgabebefehle

Ein/Ausgabebefehle sind zum Umgang mit Ein/Ausgabeeinheiten spezialisierte Befehle. In der Praxis benutzen allerdings nahezu alle Mikroprozessoren in den Speicherbereich einbezogene (memory-mapped) Ein/Ausgabeeinheiten. Das bedeutet, daß diese Einheiten gerade wie Speicherbausteine an den Adreßbus angeschlossen

und als solche behandelt werden. Sie erscheinen dem Programmierer wie Speicherstellen. Man kann alle speicherbezogenen Befehle auf sie anwenden. Das hat den Vorzug, daß eine große Zahl von Befehlen eingesetzt werden kann. Der Nachteil liegt darin, daß speicherbezogene Befehle in der Regel drei Bytes benötigen und deshalb langsam ausgeführt werden. Zum leistungsfähigen Umgang mit Ein- und Ausgabeeinheiten in einer solchen Umgebung wäre ein besonderer kurzer Adressierungsmodus wünschenswert. Man legt daher oft Ein/Ausgabeeinheiten mit besonderen Geschwindigkeitsanforderungen in Speicherseite 0. Wenn jedoch eine „Zero-Page"-Adressierung möglich ist, wird diese Speicherseite normalerweise für Schreib/Lesespeicher eingesetzt und ermöglicht so ihren Einsatz für Ein/Ausgabeeinheiten nur mit zusätzlichen Entwurfsschwierigkeiten.

Steuerbefehle

Steuerbefehle im hier gemeinten Sinn stellen Synchronisationssignale zur Verfügung und können ein Programm anhalten oder unterbrechen. Sie können außerdem eine externe Programmunterbrechung (interrupt) nachbilden. (Programmunterbrechungen werden in Kapitel 6 bei den Ein/Ausgabetechniken beschrieben).

Beim 6502 mögliche Befehle

Transferbefehle

Der 6502 bietet einen vollständigen Satz von Transferbefehlen an, wobei allerdings der Ladebefehl für den Stapelzeiger weniger flexibel als die anderen ist. Man kann den Inhalt des Akkumulators mit den Befehlen LDA aus dem Speicher laden oder mit STA dort ablegen. Dasselbe gilt für Register X und Register Y, wo die Befehle LDX bzw. LDY und STX bzw. STY heißen. Der Stapelzeiger S ist nicht direkt aus dem Speicher ladbar.
Dazu gibt es noch Befehle zum Registerverkehr: TAX und TAY übertragen (transferieren) den Akkumulatorinhalt in das X- bzw. das Y-Register. Die umgekehrte Richtung drückt TXA und TYA aus. Der Inhalt des Stapelzeigers kann nur mit dem X-Register ausgetauscht werden und zwar mit TXS (X-Inhalt in Stapel übertragen) und TSX für die Gegenrichtung.
Es gibt keinen (Zweiadreß-)Befehl zum unmittelbaren Austausch des Inhalts von Speicherstellen (z.B. „Übertrage den Inhalt von LOC1 nach LOC2"). Dies gilt auch für die anderen Befehlsklassen.

Stapeloperation

Zwei „push"- und „pull"-Befehle stehen zur Verfügung. Sie übertragen den Inhalt des Akkumulators bzw. den des Statusregisters P auf den Stapel oder zurück und setzen den Stapelzeiger S entsprechend. Auf den Stapel werden die Inhalte vermittels PHA (Akkumulator) und PHP (P-Register) übertragen, zurück mit PLA bzw. PLP.

Datenverarbeitung

Arithmetik

Die arithmetischen Befehle lauten ADC und SBC. ADC (add with carry) bezeichnet eine Addition mit Einbezug des Übertragsbits C. Es gibt keine Addition ohne Übertrag. Das macht etwas Umstand, indem es vor einer Addition das Löschen des Übertragsbits C mittels CLC erfordert. Die Subtraktion wird entsprechend unter Einbezug des Übertragsbits C durch SBC ausgeführt (subtract with carry). Es ist zu beachten, daß SBC mit dem *negierten* Wert von C arbeitet. Man muß deshalb vor einer Subtraktion das C-Bit mit SEC auf Eins setzen.

Es gibt einen speziellen Rechenmodus, der die unmittelbare Verarbeitung von BCD-Zahlen gestattet. Viele andere Mikroprozessoren bieten nur einen BCD-Befehl als Zusatzkode an. Durch die Dezimal/Binärumschaltung mittels der Dezimalflagge D verdoppelt sich beim 6502 die Anzahl der verfügbaren Arithmetikoperationen.

Inkrementieren und Dekrementieren

Inkrementieren bedeutet hier Weiterzählen eines Registerinhalts um Eins, Dekrementieren entsprechend das Herunterzählen. Man kann beim 6502 Speicherstellen sowie die Register X und Y in- bzw. dekrementieren, *nicht aber den Akkumulator*. Dabei arbeiten INC (increment memory) und DEC (decrement memory) mit dem Inhalt von Speicherstellen, während die Befehle INX, INY und DEX, DEY auf die beiden Indexregister X bzw. Y wirken.

Logische Operationen

Es stehen die drei klassischen logischen Operationen AND (UND), ORA (ODER) und EOR (EXKLUSIV-ODER) zur Verfügung. Sehen wir uns die Arbeit dieser Befehle im Einzelnen an:

AND

Jede logische Operation wird durch die eine Wahrheitstabelle charakterisiert, die den logischen Wert des Ergebnisses in Abhängigkeit des logischen Werts der beiden Operanden wiedergibt. Die Wahrheitstabelle der AND-Funktion (UND) sieht so aus:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

Die AND-Operation ergibt eine „1“ nur dann, wenn beide Eingangsoperanden auch den Wert „1“ haben. Mit anderen Worten: Hat irgendeiner der beiden Operanden den Wert „0“, dann hat auch das Ergebnis den Wert „0“. Diese Eigenschaft benutzt man, um eine Bitstelle in einem Wort gezielt auf „0“ zu setzen, was als „(Aus-)Maskieren“ bezeichnet wird.

Eine der wichtigsten Anwendungen des AND-Befehls ist damit das Löschen einer oder mehrerer Bitstellen in einem Wort. Nehmen wir z. B. an, in einem Wort müßten die vier rechten Bits auf Null gesetzt werden, die übrigen aber erhalten bleiben. Das läßt sich durch das folgende Programm erledigen:

```
LDA     WORT            WORT LADEN
AND     #%11110000      BITS 0 BIS 3 AUSBLENDEN
```

(% dient hier zur Kennzeichnung einer Dualzahl.) Nehmen wir an, WORT habe den Wert 10101010. Das Programm ergibt dann im Akkumulator den Wert 10100000.

Übung 4.1:
Schreiben Sie ein Dreizeilenprogramm, das Bits 1 und 6 der Speicherstelle WORT auf Null setzt.

Übung 4.2:
Was geschieht bei Verknüpfung mit der Maske 11111111?

ORA
Dieser Befehl bewirkt eine (inklusive) ODER-Verknüpfung des Akkumulators mit dem angegebenen Operanden. Ihr entspricht die folgende Wahrheitstabelle:

0 ORA 0 = 0
0 ORA 1 = 1
1 ORA 0 = 1
1 ORA 1 = 1

Für die ODER-Verknüpfung ist diese Regel kennzeichnend: Wenn einer der beiden Operanden den Wert „1“ hat, steht auch im Ergebnis eine „1“. Man benutzt das, um in einem Wort bestimmte Bits auf den Wert 1 zu setzen:

```
LDA     WORT
ORA     #%00001111
```

Wenn WORT beispielsweise den Inhalt 10101010 hatte, so steht im Akkumulator nach Programmabarbeitung der Wert „10101111“.

Übung 4.3:
Was geschieht, wenn wir hier den Befehl ORA #%10101111 einsetzen?

Übung 4.4:
Was für ein Ergebnis bringt die ODER-Verknüpfung mit dem hexadezimalen Wert „EE“?

EOR
EOR steht für „exclusive OR“ – EXKLUSIV-ODER. Die EXKLUSIV-ODER-Verknüpfung unterscheidet sich von der oben beschriebenen inklusiven Form nur in ei-

nem Punkt: Sie gibt eine Entweder-Oder-Beziehung wieder. Das Ergebnis ist nur dann „1“, wenn genau einer der beiden Operanden den Wert „1“ hat. Haben beide den Wert „1“, dann erhält man als Ergebnis eine „0“ anstatt einer „1“ wie beim inklusiven ODER. Die Wahrheitstabelle lautet damit:

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0

Man setzt den EOR-Befehl für Vergleiche ein. Wenn irgendein Bit der zu vergleichenden Operanden sich vom andern unterscheidet, so erhält man einen Wert ungleich Null als Ergebnis. Außerdem verwendet man EOR beim 6502 zum *Komplementieren* eines Worts, da es hierfür keinen eigenen Befehl gibt. Das geschieht, indem man das Wort mit lauter Einsen verknüpft. Folgendes Programm läßt sich dazu verwenden:

```
LDA     WORT
EOR     #%00001111
```

Nehmen wir an, daß WORT den Wert 10101010 hatte. Dann steht nach Programmabarbeitung im Akkumulator der Wert 01010101. Das ist das Komplement des ursprünglichen Werts.

Übung 4.5:
Was geschieht durch den Befehl EOR # $00?

Schiebeoperationen

Es gibt zwei Schiebeoperationen beim 6502: ASL (arithmetic shift left), der den Registerinhalt eine Stelle nach links schiebt, und LSR (logical shift right), für die entgegengesetzte Richtung. In beiden Fällen kommt das herausgeschobene Bit in die Übertragsflagge C, und auf der anderen Seite wird eine „0“ nachgeschoben.
Das wird durch die beiden Rotierbefehle ROL (rotate left) und ROR (rotate right) ergänzt, die den Registerinhalt und den Wert des Übertragsbits nach links bzw. rechts rotieren lassen.

Achtung:
Ältere Versionen des 6502-Prozessors kennen von den Rotierbefehlen nur die Linksrotation ROL! Prüfen Sie das im Zweifelsfall im Datenblatt des Herstellers Ihres Exemplars nach.

Vergleichsbefehle

Die Inhalte des Akkumulators A und der beiden Indexregister X und Y lassen sich mit denen von Speicherstellen mittels der Befehle CMP (compare), CPX und CPY vergleichen.

Tests und Verzweigungen

Da sich Tests fast ausschließlich auf die Flaggen im Statusregister P beziehen, wollen wir die beim 6502 verfügbaren Flaggen untersuchen. Der Inhalt des Statusregisters ist in Bild 4-2 wiedergegeben. Sehen wir uns die Flagge von links nach rechts an:

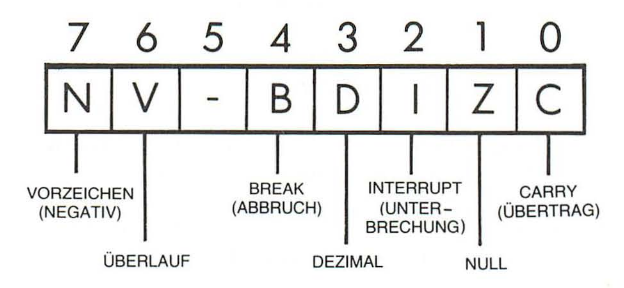


Bild 4-2: Die Flaggen im P-Register

N-Flagge (negative)
Hier wird der Wert des höchstwertigen Bits hineinkopiert, d.h. die N-Flagge gibt das *Vorzeichen* des betrachteten Bytes wieder: Ist die Zahl negativ, so hat N den Wert „1", sonst „0". Das N-Bit wird von allen Transferbefehlen und den Befehlen zur Datenverarbeitung beeinflußt.
In den meisten Fällen hat N denselben Wert wie Bit 7 des Akkumulators. Damit ist Bit 7 das einzige Bit, das bequem mit einem einzigen Befehl getestet werden kann. Alle anderen Akkumulatorbits müssen zum Test erst verschoben werden. Deshalb ist Bit 7 in allen Fällen, in denen der Inhalt eines Worts rasch untersucht werden soll, vorzuziehen. Hier liegt der Grund, weshalb die Statusbits externer Ein- bzw. Ausgabeeinheiten normalerweise über Leitung 7 des Datenbusses übertragen werden. Will man den Zustand einer solchen E/A-Einheit wissen, so liest man einfach das betreffende externe Statusregister in den Akkumulator und testet dann Bit N im Prozessorstatusregister P.
Die Befehle, die N beeinflussen, lauten: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA und TYA.

V-Flagge (overflow)
Die Aufgabe dieser Überlaufsflagge ist bereits in Kapitel 1 und 3 bei den arithmetischen Operationen diskutiert worden. Sie zeigt an, daß das Ergebnis einer Zweierkomplementsoperation wegen eines Überlaufs von Bit 6 nach Bit 7 (das Vorzeichenbit) wahrscheinlich falsch ist. Man muß ein besonderes Korrekturprogramm einsetzen, wenn dieses Bit gesetzt ist. Wenn man reine Dualzahlen, d.h. kein Zweierkomplement einsetzt, dann ist das V-Bit einem Übertrag von Bit 6 nach Bit 7 äquivalent.

Die V-Flagge wird vom BIT-Befehl noch besonders beeinflußt. Im Ergebnis dieses Befehls erhält die Flagge den Wert von Bit 6 des getesteten Datenbits.
Die V-Flagge wird von den folgenden Befehlen beeinflußt: ADC, BIT, CLV, PLP, RTI und SBC.

B-Flagge (break)
Die B-Flagge wird automatisch vom Prozessor gesetzt, wenn durch einen BRK-Befehl das Programm unterbrochen worden ist. Sie gestattet die Unterscheidung zwischen einer externen (Hardware-)Unterbrechung und einer durch BRK ausgelösten internen (Software-)Unterbrechung. Die Flagge bleibt von allen übrigen Befehlen außer PLP und RTI unberührt.

D-Flagge (decimal)
Der Einsatz dieser Flagge wurde bereits in Kapitel 3 bei der Besprechung arithmetischer Programme untersucht. Immer wenn D auf „1“ gesetzt ist, arbeitet der Prozessor im *Dezimalmodus,* in dem BCD-Rechnungen ausgeführt werden. Steht D auf „0“, so wird *binär* gerechnet. Folgende vier Befehle beeinflussen den Wert von D: CLD, PLP, RTI und SED.

I-Flagge (interrupt)
Diese Flagge gibt den Stand der *Unterbrechungsmaske* (interrupt-mask) wieder und kann vom Programmierer durch CLI oder PLP bzw. vom Mikroprozessor während eines Rücksetzens des Systems (reset) oder während einer Programmunterbrechung (interrupt) verändert werden. Ist I auf „1“ gesetzt, so ist jede weitere Programmunterbrechung blockiert („ausmaskiert“).
Folgende Befehle beeinflussen den Wert der I-Flagge: BRK, CLI, PLP, RTI und SEI.

Z-Flagge (zero)
Ist die Z-Flagge gesetzt (d.h. gleich „1“), so wird angezeigt, daß das Ergebnis eines Datenübertrags oder sonstiger Operationen den Wert Null ergeben hat. Außerdem setzt der Vergleichsbefehl die Z-Flagge auf „1“, wenn die bearbeiteten Werte gleich sind. Es gibt allerdings keinen Befehl, mit dem die Z-Flagge unmittelbar gesetzt oder gelöscht werden kann. Man erreicht dies jedoch beispielsweise auch durch folgenden Befehl:

```
LDA        #0
```

Das Z-Bit wird von vielen Befehlen beeinflußt: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, ROR, RTI, SBC, TAX, TAY, TSX, TXA und TYA.

C-Flagge (carry)
Wir haben gesehen, daß die Übertragsflagge C zwei Aufgaben erfüllt. Zum einen soll sie bei Rechnungen einen Übertrag anzeigen (oder die Tatsache, daß vom nächsthöheren Byte ein Bit geborgt worden ist). Zum anderen nimmt sie das bei einer Schiebe- oder Rotieroperation aus dem Register hinausgeschobene Bit auf. Diese beiden Aufgaben müssen nicht unbedingt mit einer Flagge kombiniert werden und werden es bei größeren Computern auch nicht. Jedoch spart dieser Ansatz bei Mikroprozessoren Zeit ein, insbesondere bei Multiplikations- und Divisionsprogrammen. Das Übertragsbit läßt sich durch SEC und CLC ausdrücklich setzen oder löschen.
Folgende Befehle beeinflussen die C-Flagge: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, ROR, RTI, SBC und SEC.

Test- und Verzweigungsbefehle

Man kann nicht alle 6502-Flaggen im P-Register unmittelbar auf ihren Wert untersuchen. Nur vier der sieben Bits lassen sich testen, und folglich gibt es acht Verzweigungsbefehle („branch"-Befehle). Sie lauten:

- BMI (branch on minus) und BPL (branch on plus). Diese beiden Befehle testen die N-Flagge. BMI verzweigt bei N = 1, BPL bei N = 0.
- BCC (branch on carry clear) und BCS (branch on carry set) testen die Übertragsflagge C. BCC verzweigt bei C = 0 und BCS bei C = 1.
- BEQ (branch if equal) und BNE (branch if not equal) beruhen auf der Tatsache, daß der Vergleichsbefehl CMP die Z-Flagge setzt, wenn Gleichheit vorliegt. So verzweigt BEQ (bei Gleichheit), wenn Z = 1 ist; BNE (bei Ungleichheit) verzweigt Z = 0.
- BVS (branch on overflow set) und BVC (branch on overflow clear) beziehen sich auf die Übertragsflagge V. BVS verzweigt bei V = 1, BVC bei V = 0.

Diese acht Befehle testen und verzweigen im selben Arbeitsgang. Dabei wird das Sprungziel durch den Abstand (displacement) vom gerade gegebenen Befehl (dem Programmzählerstand) in acht Bits bestimmt. Das erlaubt Sprünge über – 128 Bytes (nach hinten) bzw. + 127 Bytes (nach vorne). Diese Abstandsangabe wird zu der Adresse des auf den Verzweigungsbefehl folgenden Befehls addiert (d.h. zu dem Stand des Programmzählers nach Übernahme desrgibt das maximale Sprungweiten von – 128 + 2 = – 126 nach hinten und + 127 + 2 = 129 nach vorne.
Es gibt noch zwei weitere Sprungbefehle, die nicht an eine Sprungbedingung gebunden sind, d. h. immer ausgeführt werden: JMP (jump) und JSR ist ein Sprung zu einer 16-Bit-Adresse. JSR ist der Unterprogrammaufruf. Er springt zu der neuen Adresse und rettet gleichzeitig den alten Programmzählerstand auf den Stapel. Da diese beiden Befehle keinen Bedingungen unterliegen, werden sie in der Regel einem der oben dargestellten „branch"-Verzweigungsbefehle, die immer eine Flagge testen, nachgestellt.

Es gibt zwei Rücksprungbefehle: RTS (return from subroutine) und RTI (return from interrupt). RTI wirt im Abschnitt über Programmunterbrechungen (interrupts) betrachtet, RTS ist ein Rücksprung von einem Unterprogramm und holt die auf der Stapelspitze abgelegte Adresse in den Programmzähler (wobei gleichzeitig der Stapelzeiger S um 2 weitergezählt wird).
Zwei besondere Befehle dienen diesem Test von Bits in Registern und dem Größenvergleich der Registerinhalte.
Der BIT-Befehl führt eine UND-Verknüpfung zwischen Akkumulator und Speicherinhalten durch, *ändert dabei jedoch den Akkumulatorinhalt nicht.* Die N-Flagge erhält den Wert von Bit 7 der getesteten Speicherstelle, die V-Flagge den von Bit 6. Das Ergebnis der UND-Verknüpfung wird in der Z-Flagge festgehalten, die auf „1" gesetzt wird, wenn ein Nullbyte herauskam. Üblicherweise lädt man eine Maske in den Akkumulator und testet mit Hilfe des BIT-Befehls aufeinanderfolgende Speicherstellen. Wenn die Maske z. B. nur eine einzige „1" enthält, so wird dadurch getestet, ob in dem betreffenden Speicherwort dieses Bit gesetzt (Z = 1) oder gelöscht (Z = 0) ist.

Da Bit 6 und 7 des getesteten Bytes automatisch in V bzw. in N übertragen werden, benutzt man die Maske in der Praxis meist nur zum Testen von Bit 0 bis 5.
Der CMP-Befehl (compare) vergleicht den Inhalt der adressierten Speicherstelle mit dem Akkumulator durch eine Subtraktionsoperation, bei der aber der ursprüngliche Akkumulatorinhalt erhalten bleibt. Das Ergebnis wird in den Flaggen N und Z festgehalten, mit deren Hilfe man Gleichheit (Z = 1), Speicher größer als Akkumulator (Z = 0, N = 1) und Speicher kleiner als Akkumulator (Z = 0, N = 0) feststellen kann. CPX und CPY führen dieselbe Operation mit den Indexregistern anstelle des Akkumulators als Bezug aus.
Die von den „branch"-Befehlen nicht erfaßten Bits lassen sich im übrigen nur dadurch testen, daß man sie in den Akkumulator bringt, was am besten über den Stapel geschieht:

```
PHP          P-REGISTER AUF DEN STAPEL BRINGEN
PLA          STAPELSPITZE IN DEN AKKUMULATOR HOLEN
```

Dann steht die I-Flagge in Bit 2 des Akkumulators, die D-Flagge in Bit 3 und die B-Flagge in Bit 4.

Ein/Ausgabebefehle

Es gibt keine besonderen Ein- oder Ausgabebefehle beim 6502.

Steuerbefehle

Die Steuerbefehle umfassen unter anderem die Befehle, mit denen verschiedene Flaggen unmittelbar beeinflußt werden können. Es sind die Löschbefehle CLC, CLD, CLI und CLV für die Flaggen C, D, I und V sowie die Setzbefehle SEC, SED, SEI und SEV für diese Flaggen.
Der BRK-Befehl führt eine Software-Unterbrechung durch und wird in Kapitel 7 im Abschnitt über Programmunterbrechungen (interrupts) behandelt.
Der NOP-Befehl schließlich (no operation) ist ein Befehl, der nichts tut. Man setzt ihn gewöhnlich zum Verlängern von Verzögerungsschleifen ein oder als Platzhalter für später einzufügende Befehle.
Sehen wir uns jetzt die verschiedenen Befehle im einzelnen an. Dabei sei empfohlen, die Darstellung zunächst einmal nur zu überfliegen und dann sofort den Abschnitt über Adressierungstechniken durchzuarbeiten.

Teil 2: Die Befehle

Abkürzungen

A	Akkumulator
C	Übertrag (carry)
M	Speicheradresse (memory)
P	Statusregister
S	Stapelzeiger
X	Indexregister X
Y	Indexregister Y
ADR	Adresse
DATEN	Adressierte Daten
DISP	Abstand (displacement)
HEX	Hexadezimal
PC	Programmzähler
PCH	Programmzähler, höherwertig (high)
PCL	Programmzähler, niederwertig (low)
STAPEL	Inhalt der Stapelspitze
V	logisches ODER
$\wedge$	logisches UND
$\veebar$	logisches EXKLUSIV-ODER
●	ändert
$\leftarrow$	Zuweisung (erhält den Wert)
( )	Inhalt von
(M6)	Bitstelle 6 unter Adresse M
b	Einzelbit, besonders angegeben

ADC

Add with carry
Mit Übertrag addieren

Funktion: A ← (A) + DATEN + C

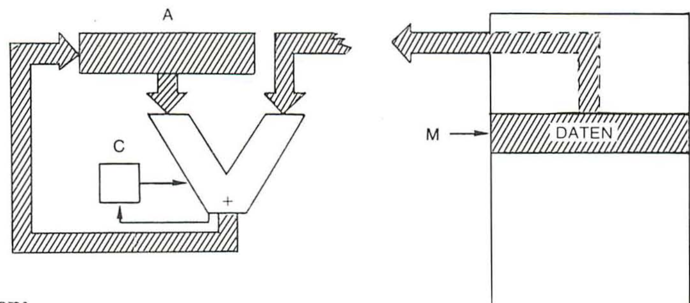
Format:

011bbb01	ADR/DATEN	ADR

Beschreibung:
Addiert den Inhalt der angegebenen Speicheradresse oder Konstanten zum Akkumulator plus dem Wert des Übertragsbits C. Das Ergebnis steht im Akkumulator, der Übertrag in C.

Besonderheiten:
- ADC kann sowohl dezimal als auch binär arbeiten. Der Arbeitsmodus wird durch die Dezimalflagge D bestimmt, die vor der Addition auf den in Frage kommenden Wert gesetzt sein muß.
- Um ohne Übertrag zu addieren, muß vor der Rechnung das Übertragsbit C mit CLC gelöscht werden.


Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			6D	65	69	7D	79	61	71	75			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●	●				●	●

Befehlskodes:

Absolut:	01101101 bbb = 011	16-Bit-Adresse HEX = 6D 4 Taktzyklen
Zero-Page:	01100101 bbb = 001	8-Bit-Adresse HEX = 65 3 Taktzyklen
Unmittelbar:	01101001 bbb = 010	Daten HEX = 69 2 Taktzyklen
Absolut, X:	01111101 bbb = 111	16-Bit-Adresse HEX = 7D 4 Taktzyklen*
Absolut, Y:	01111001 bbb = 110	16-Bit-Adresse HEX = 79 4 Taktzyklen*
(Indirekt, X):	01100001 bbb = 000	8-Bit-Adresse HEX = 61 6 Taktzyklen
(Indirekt, Y):	01110001 bbb = 100	8-Bit-Adresse HEX = 71 5 Taktzyklen*
Zero-Page, X:	01110101 bbb = 101	8-Bit-Adresse HEX = 75 4 Taktzyklen

*: plus 1 Taktzyklus bei Seitenüberschreitung

AND

Logisches UND

Funktion: A ← (A) ∧ DATEN

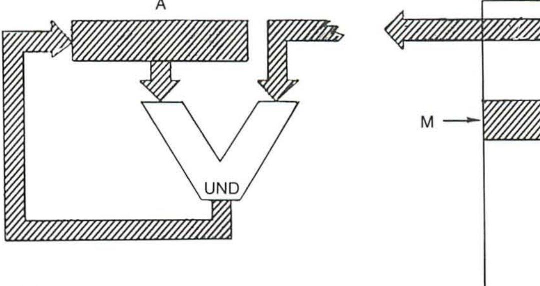
Format:

011bbb01	ADR/DATEN	ADR

Beschreibung:
UND-verknüpft den Akkumulatorinhalt bitweise mit den angegebenen Daten. Das Ergebnis steht im Akkumulator. Die Verknüpfung erfolgt nach der Tabelle:

A\M	0	1
0	0	0
1	0	1

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			2D	25	29	3D	39	21	31	35			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	00101101 bbb = 011	16-Bit-Adresse HEX = 2D 4 Taktzyklen
Zero-Page:	00100101 bbb = 001	8-Bit-Adresse HEX = 25 3 Taktzyklen
Unmittelbar:	00101001 bbb = 010	Daten HEX = 29 2 Taktzyklen
Absolut, X:	00111101 bbb = 111	16-Bit-Adresse HEX = 3D 4 Taktzyklen*
Absolut, Y:	00111001 bbb = 110	16-Bit-Adresse HEX = 39 4 Taktzyklen*
(Indirekt, X):	00100001 bbb = 000	8-Bit-Adresse HEX = 21 6 Taktzyklen
(Indirekt, Y):	00110001 bbb = 100	8-Bit-Adresse HEX = 31 5 Taktzyklen*
Zero-Page, X:	00110101 bbb = 101	8-Bit-Adresse HEX = 35 4 Taktzyklen*

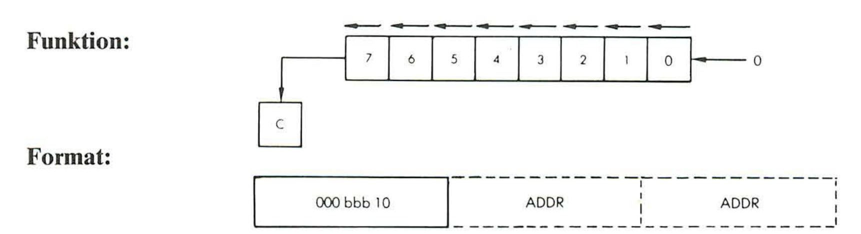
*: plus 1 Taktzyklus bei Seitenüberschreitung

ASL

arithmetic shift left
Arithmetisch links schieben

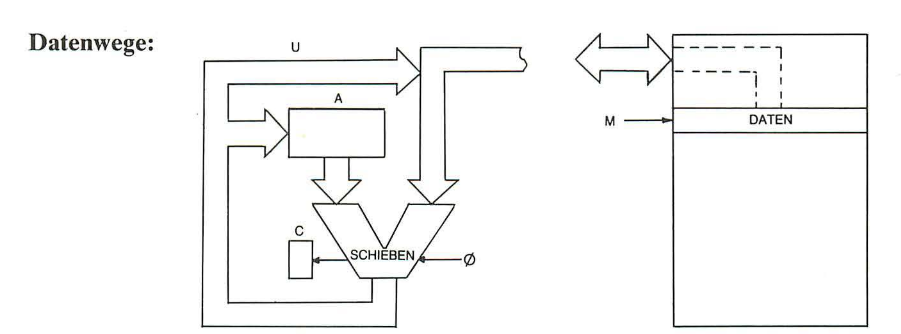
Funktion:

Format:



Beschreibung:
Akkumulator oder Speicherstelleninhalt um eine Bitstelle nach links schieben. Bit 0 wird zu „0“, Bit 7 wird in das Übertragsbit C geschoben. Das Ergebnis steht in der Datenquelle (Akkumulator oder Speicherstelle).

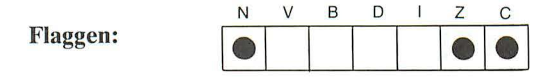
Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX		0A	0E	06		1E				16			
BYTES		1	3	2		3				2			
ZYKLUS		2	6	5		7				6			
bbb		010	011	001		111				101			

Flaggen:



Befehlskodes:

Akkumulator:	00001010	
	bbb = 010	HEX = 0A 2 Taktzyklen
Absolut:	00001110	16-Bit-Adresse
	bbb = 011	HEX = 0E 6 Taktzyklen
Zero-Page:	00000110	8-Bit-Adresse
	bbb = 001	HEX = 06 5 Taktzyklen
Absolut, X:	00011110	16-Bit-Adresse
	bbb = 111	HEX = 1E 7 Taktzyklen
Zero-Page, Y:	00010110	8-Bit-Adresse
	bbb = 101	HEX = 16 6 Taktzyklen

BCC

branch on carry clear
Verzweigen bei gelöschtem Übertrag

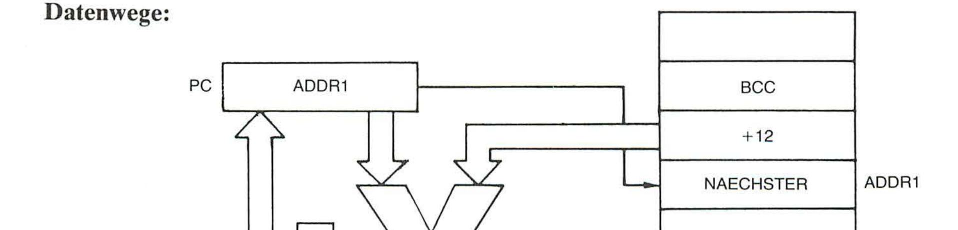
Funktion: Gehe zur angegebenen Adresse, falls C = 0 ist.

Format:

10010000	DISP

Beschreibung:
Testet die Übertragsflagge C. Ist C = 0, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist C = 1, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BCC-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = 90, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BCS

branch on carry set
Verzweigen bei gesetztem Übertrag

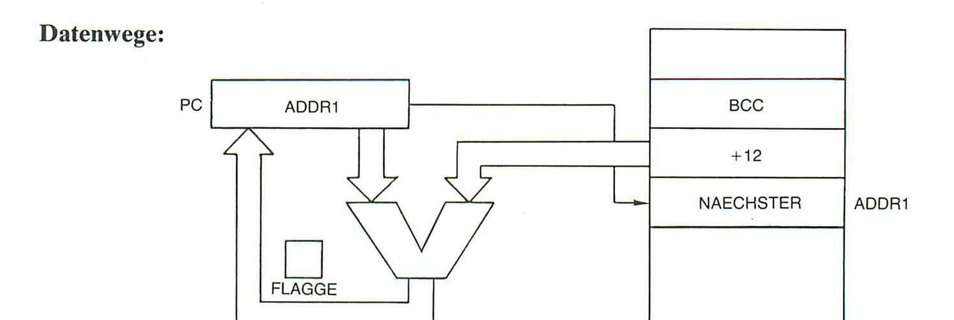
Funktion: Gehe zur angegebenen Adresse, falls C = 1 ist.

Format:

10110000	DISP

Beschreibung:
Testet die Übertragsflagge C. Ist C = 1, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist C = 0, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BCS-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = B0, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BEQ

branch on equal to zero
Verzweigen falls gleich Null

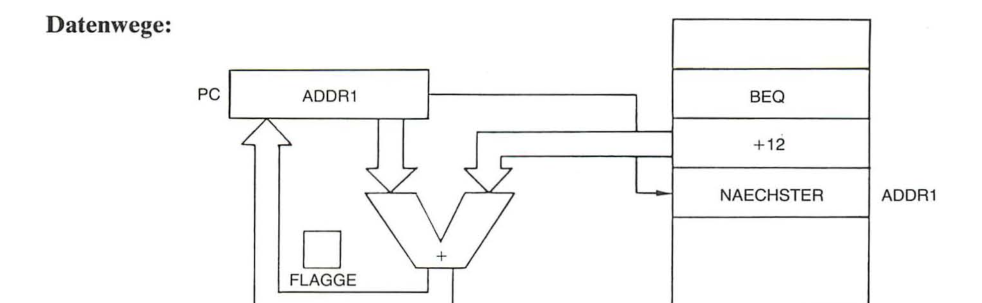
Funktion: Gehe zur angegebenen Adresse, falls Z = 1 ist.

Format:

11110000	DISP

Beschreibung:
Testet die Nullflagge Z. Ist Z = 1, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist Z = 0, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BEQ-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = F0, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BIT

Speicherbits testen

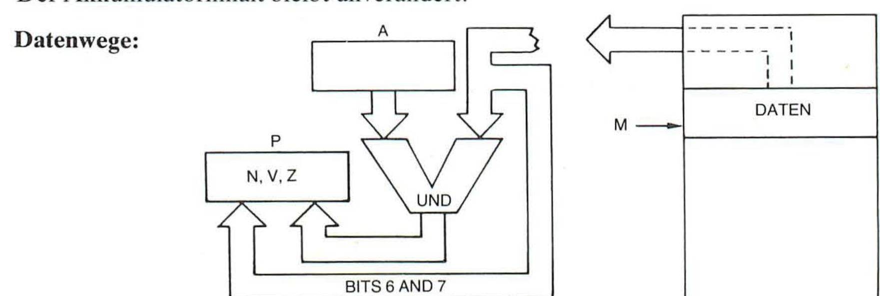
Funktion: Z ← (A) ∧ (M), N ← (M7), V ← (M6)

Format:

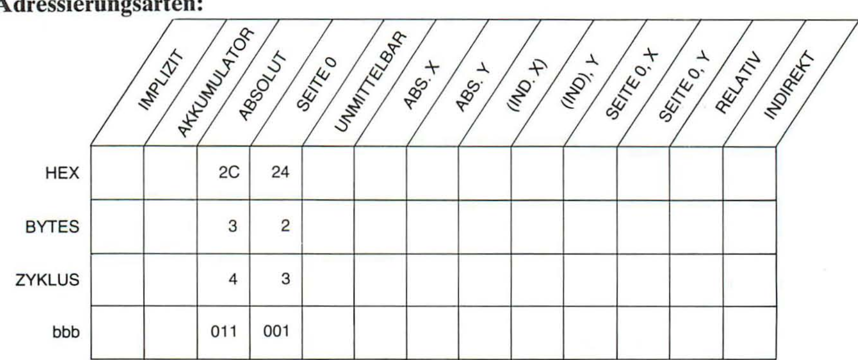
0010b100	ADR	ADR

Beschreibung:
Die Inhalte von A und M werden UND-verknüpft, das resultierende Byte jedoch nicht festgehalten. Das Vergleichsergebnis steht in der Z-Flagge: Z = 0, wenn alle getesteten Bits gelöscht (= „0“) sind; Z = 1 sonst. Zusätzlich werden Bits 6 und 7 der adressierten Speicherstelle in die Flagge V bzw. N im Statusregister P übertragen. Der Akkumulatorinhalt bleibt unverändert.

Datenwege:



Adressierungsarten:



	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			2C	24									
BYTES			3	2									
ZYKLUS			4	3									
bbb			011	001									

Flaggen:

N	V	B	D	I	Z	C
M_7	M_6				●	

Befehlskodes:

Absolut:	00101100 b = 1	16-Bit-Adresse HEX = 2C 4 Taktzyklen
Zero-Page:	00100100 b = 0	8-Bit-Adresse HEX = 24 3 Taktzyklen

BMI

branch on minus
Verzweigen falls negativ

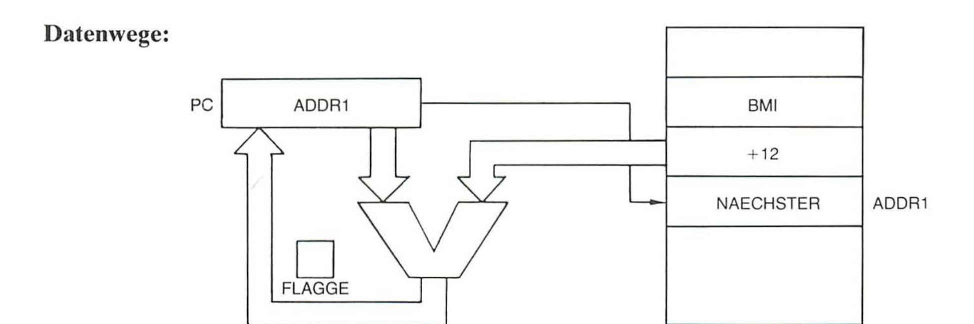
Funktion: Gehe zur angegebenen Adresse, falls N = 1 ist. (Ergebnis < 0).

Format:

00110000	DISP

Beschreibung:
Testet die Vorzeichenflagge N. Ist N = 1, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist N = 0, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BMI-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = 30, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BNE

branch on not equal to zero
Verzweigen falls ungleich Null

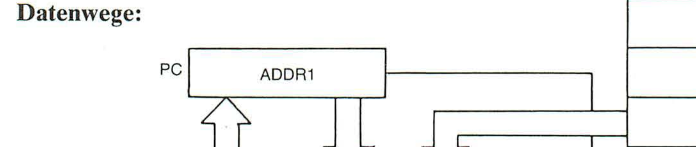
Funktion: Gehe zur angegebenen Adresse, falls Z = 0 ist. (Ergebnis #0)

Format:

11010000	DISP

Beschreibung:
Testet die Nullflagge Z. Ist Z = 0, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist Z = 1, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BCS-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = D0, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BPL

branch on plus
Verzweigen falls positiv

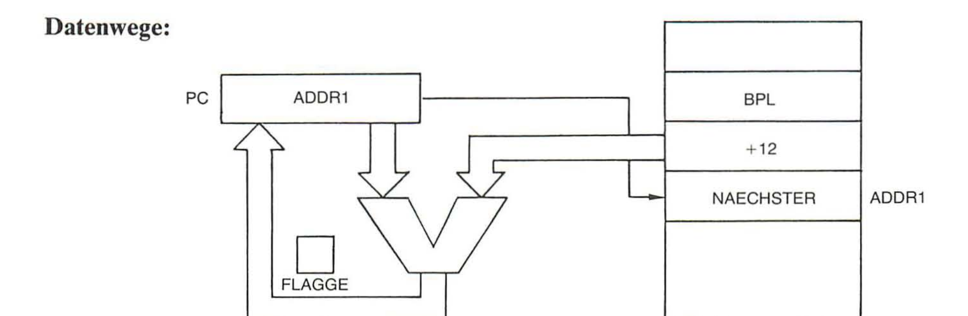
Funktion: Gehe zur angegebenen Adresse, falls N = 0 ist. (Ergebnis ≥ 0)

Format:

00010000	DISP

Beschreibung:
Testet die Vorzeichenflagge N. Ist N = 0, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist N = 1, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BPL-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = 10, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BRK

break
Unterbrechung durch Software

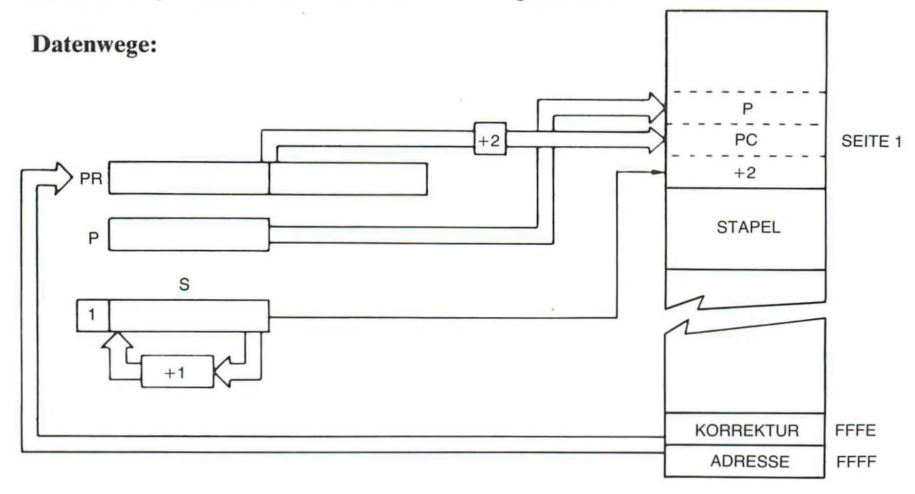
Funktion: STAPEL (PC) + 2, STAPEL (P), PC ← (FFFE, FFFF)

Format: 00000000

Beschreibung:
Arbeitet wie eine Programmunterbrechung: Der Wert des Programmzählers PC wird auf den Stapel gebracht, danach der Inhalt des Statusregisters P. Der Inhalt der Speicherstellen FFFE und FFFF (hexadezimal) wird darauf in den Programmzähler übertragen: (FFFE) kommt in den niederwertigen Teil (PCL), (FFFF) kommt in den höherwertigen Teil (PCH). Der auf den Stapel gebrachte P-Inhalt hat die B-Flagge auf „1" gesetzt, um so BRK von IRQ zu unterscheiden.

Wichtig:
Anders als im Fall der Unterbrechung durch Hardware wird (PC) + 2 gespeichert. Das wird u. U. nicht auf den nächsten Befehl zeigen, so daß eine Korrektur notwendig werden kann. Man macht das wegen des üblichen Einsatzes von BRK als Ersatz eines Zweibytebefehls beim Austesten von Programmen.

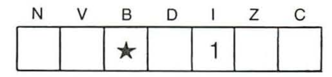
Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 00, 1 Byte, 7 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
		★		1		



Bau: B wird gesetzt, bevor P auf den Stapel kommt

BVC

branch on overflow clear
Verzweigung falls kein Überlauf

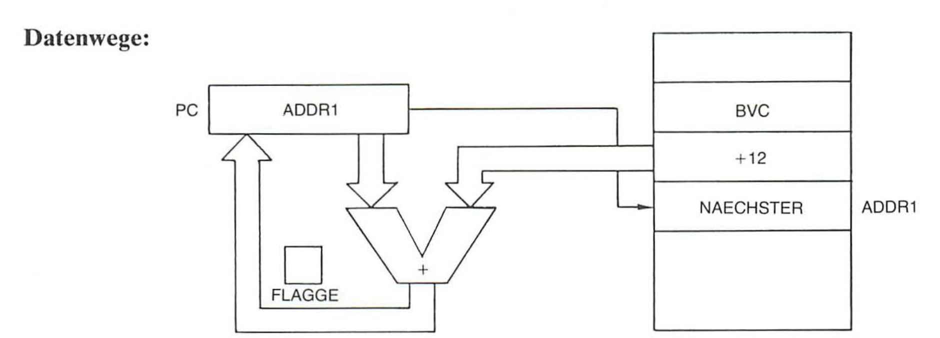
Funktion: Gehe zur angegebenen Adresse, falls V = 0 ist.

Format:

01010000	DISP

Beschreibung:
Testet die Überlaufsflagge V. Ist V = 0, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist V = 1, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von –126 bis +129 um den BVC-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = 50, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



BVS

branch on overflow set
Verzweigen falls Überlauf

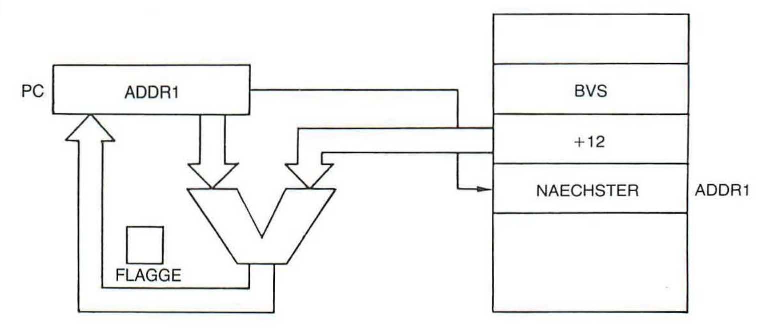
Funktion: Gehe zur angegebenen Adresse, falls V = 1 ist.

Format:

01110000	DISP

Beschreibung:
Testet die Überlaufsflagge V. Ist V = 1, dann verzweigt die Abarbeitung zur nächsten Adresse plus dem angegebenen Abstand (in Zweierkomplementform: –128 bis +127). Ist V = 0, so erfolgt keine besondere Aktion. Da die Abstandsangabe zur Adresse des *folgenden* Befehls addiert wird, läßt sich insgesamt ein Raum von −126 bis +129 um den BPL-Befehl herum überspringen.

Datenwege:



Adressierungsarten:
Nur Relativadressierung möglich:
HEX = 70, 2 Bytes, 2 Taktzyklen + 1, wenn Verzweigung erfolgt,
+ 2, wenn die Seitengrenze überschritten wird.

Flaggen:



CLC

clear carry
Übertragsflagge löschen

Funktion: C ← Ø

Format: 00011000

Beschreibung:
Die Übertragsflagge C wird auf 0 gesetzt. Das ist oft vor einem ADC-Befehl notwendig.

Adressierungsarten:
Nur implizit:
HEX = 18, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
						Ø

CLD

clear dezimal mode
Dezimalmodus abschalten

Funktion: D ← Ø

Format: 11011000

Beschreibung:
Die Dezimalflagge D wird auf 0 gesetzt. Damit werden die nachfolgenden ADC- und SBC-Befehle binär durchgeführt, bis die Dezimalflagge wieder auf 1 gesetzt worden ist (durch SED).

Adressierungsarten:
Nur implizit:
HEX = D8, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
			Ø			

CLI

clear interrupt mask
Unterbrechungsmaske löschen

Funktion: I ← ∅

Format: 01011000

Beschreibung:
Die Unterbrechungsmaske I im Statusregister P wird auf 0 gesetzt. Dadurch werden weitere Programmunterbrechungen möglich gemacht. Ein Programm zur Unterbrechungsverarbeitung muß die I-Flagge immer löschen, soll keine folgende Unterbrechungsanforderung verlorengehen.

Adressierungsarten:
Nur implizit:
HEX = 58, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
				∅		

CLV

clear overflow flag
Überlaufsflagge löschen

Funktion: V ← ∅

Format: 10111000

Beschreibung:
Die Überlaufsflagge V wird auf 0 gesetzt.

Adressierungsarten:
Nur implizit:
HEX = B8, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
	∅					

CMP

compare to accumulator
Mit Akkumulator vergleichen

Funktion: N, Z, C ← (A) – DATEN

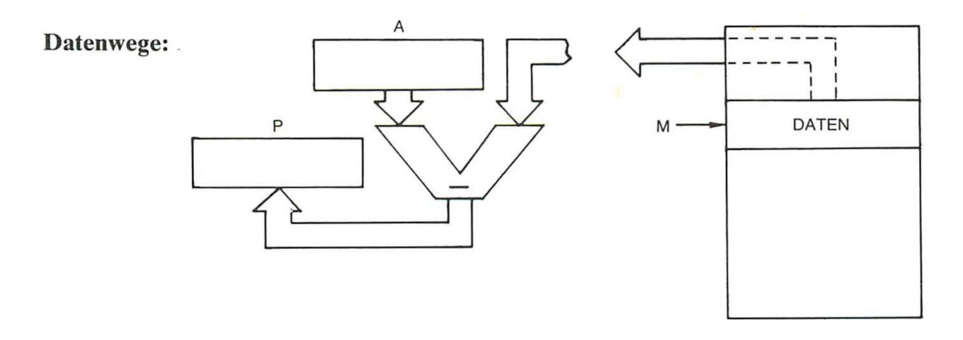
	(A) < DATEN	(A) = DATEN	(A) > DATEN
N:	0	0	1
Z:	0	1	0
C:	1	1	0

Format:

110bbb01	ADR/DATEN	ADR

Beschreibung:
Die adressierten Daten werden von Register A abgezogen, das Ergebnis jedoch nicht gespeichert. Es werden lediglich die drei Flaggen N, Z und C dem Ergebnis entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1, wenn der Inhalt von A kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt von A kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt von A größer oder gleich den angegebenen Daten ist.
Üblicherweise wird der Vergleichsbefehl von einer Verzweigung gefolgt. Dabei entdeckt BEQ Gleichheit, BNE Ungleichheit, BMI oder BCC die Kleiner-als-Bedingung und BEQ gefolgt von BCS oder BPL die Größer-als-Bedingung. BCS und BPL verzweigen im Falle „größer oder gleich“ und BEQ gefolgt von BCC oder BMI entdeckt „kleiner oder gleich“.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			CD	C5	C9	DD	D9	C1	D1	D5			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	●

Befehlskodes:

Absolut:	11001101 bbb = 011	16-Bit-Adresse HEX = CD 4 Taktzyklen
Zero-Page:	11000101 bbb = 001	8-Bit-Adresse HEX = C5 3 Taktzyklen
Unmittelbar:	11001001 bbb = 010	Daten HEX = C9 2 Taktzyklen
Absolut, X:	11011101 bbb = 111	16-Bit-Adresse HEX = DD 4 Taktzyklen*
Absolut, Y:	11011001 bbb = 110	16-Bit-Adresse HEX = D9 4 Taktzyklen*
(Indirekt, X):	11000001 bbb = 000	8-Bit-Adresse HEX = C1 6 Taktzyklen
(Indirekt, Y):	11010001 bbb = 100	8-Bit-Adresse HEX = D1 5 Taktzyklen*
Zero-Page, X:	11010101 bbb = 101	8-Bit-Adresse HEX = D5 4 Taktzyklen

*: plus 1 Taktzyklus bei Seitenüberschreitung

CPX

compare to register X
Mit Register X vergleichen

Funktion: N, Z, C ← (X) – DATEN

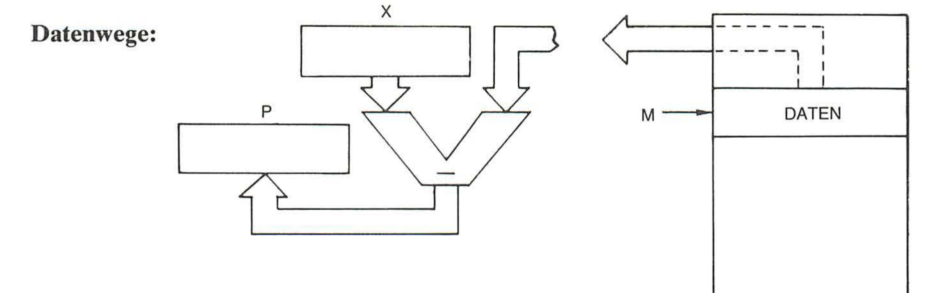
	(X) > DATEN	(X) = DATEN	(X) < DATEN
N:	0	0	1
Z:	0	1	0
C:	1	1	0

Format:

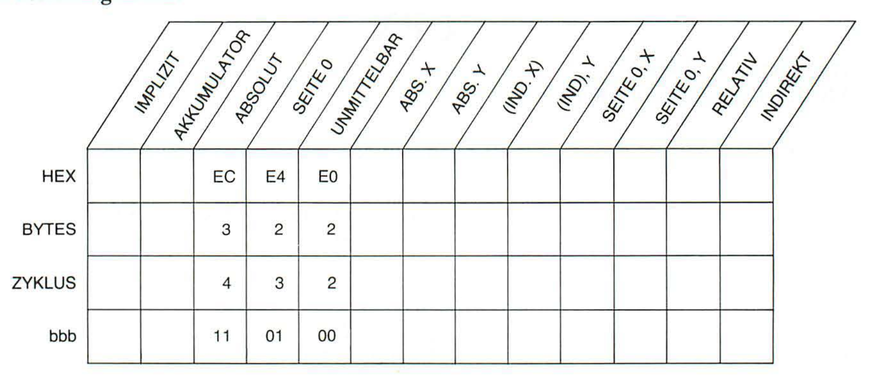
1110bb00	ADR/DATEN	ADR

Beschreibung:
Die adressierten Daten werden von Register X abgezogen, das Ergebnis jedoch nicht gespeichert. Es werden lediglich die drei Flaggen N, Z und C dem Ergebnis entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1, wenn der Inhalt von A kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt von A größer oder gleich den angegebenen Daten ist.
Üblicherweise wird der Vergleichsbefehl von einer Verzweigung gefolgt. Dabei entdeckt BEQ Gleichheit, BNE Ungleichheit, BMI oder BCC die Kleiner-als-Bedingung und BEQ gefolgt von BCS oder BPL die Größer-als-Bedingung. BCS und BPL verzweigen im Falle „größer oder gleich" und BEQ gefolgt von BCC oder BMI entdeckt „kleiner oder gleich".

Datenwege:



Adressierungsarten:



	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			EC	E4	E0								
BYTES			3	2	2								
ZYKLUS			4	3	2								
bbb			11	01	00								

Flaggen:

N	V	B	D	I	Z	C
●					●	●

Befehlskodes:

Absolut:	11101100 bbb = 11	16-Bit-Adresse HEX = EC 4 Taktzyklen
Zero-Page:	11100100 bbb = 01	8-Bit-Adresse HEX = E4 3 Taktzyklen
Unmittelbar:	11100000 bbb = 00	Daten HEX = E0 2 Taktzyklen

CPY

compare to register Y
Mit Register Y vergleichen

Funktion: N, Z, C ← (Y) – DATEN

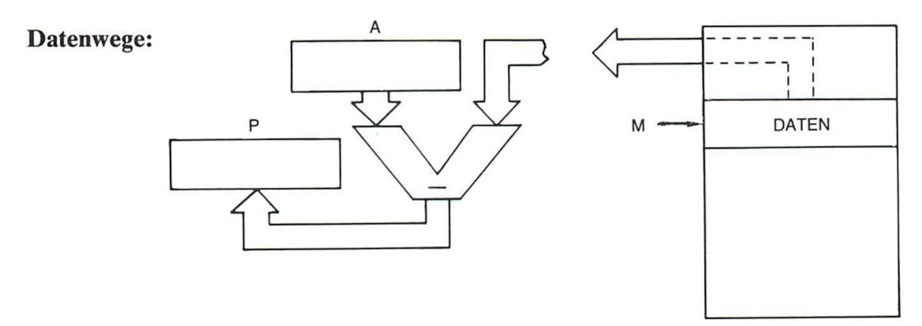
	(Y) > DATEN	(Y) = DATEN	(Y) < DATEN
N:	0	0	1
Z:	0	1	0
C:	1	1	0

Format:

1100bb00	ADR/DATEN	ADR

Beschreibung:
Die adressierten Daten werden von Register Y abgezogen, das Ergebnis jedoch nicht gespeichert. Es werden lediglich die drei Flaggen N, Z und C dem Ergebnis entsprechend gesetzt. Z = 1, wenn beide Werte gleich sind. N = 1, wenn der Inhalt von A kleiner als die adressierten Daten ist. C = 1, wenn der Inhalt von A größer oder gleich den angegebenen Daten ist.
Üblicherweise wird der Vergleichsbefehl von einer Verzweigung gefolgt. Dabei entdeckt BEQ Gleichheit, BNE Ungleichheit, BMI oder BCC die Kleiner-als-Bedingung und BEQ gefolgt von BCS oder BPL die Größer-als-Bedingung. BCS und BPL verzweigen im Falle „größer oder gleich" und BEQ gefolgt von BCC oder BMI entdeckt „kleiner oder gleich".

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			CC	C4	C0								
BYTES			3	2	2								
ZYKLUS			4	3	2								
bbb			11	01	00								

Flaggen:

N	V	B	D	I	Z	C
●					●	●

Befehlskodes:

Absolut:	11001100 bb = 11	16-Bit-Adresse HEX = CC 4 Taktzyklen
Zero-Page:	11000100 bb = 01	8-Bit-Adresse HEX = C4 3 Taktzyklen
Unmittelbar:	11000000 bb = 00	Daten HEX = C0 2 Taktzyklen

DEC

decrement memory
Speicher dekrementieren

Funktion: M ← (M) − 1

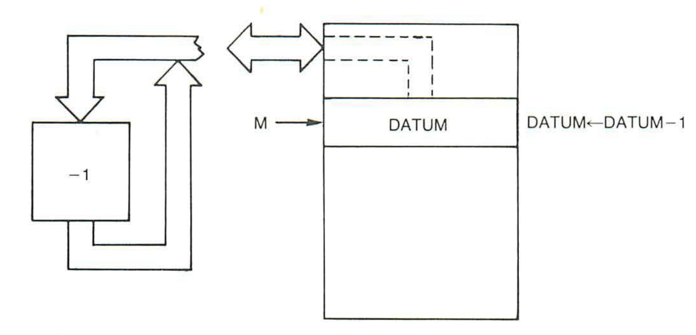
Format:

110bb110	ADR	ADR

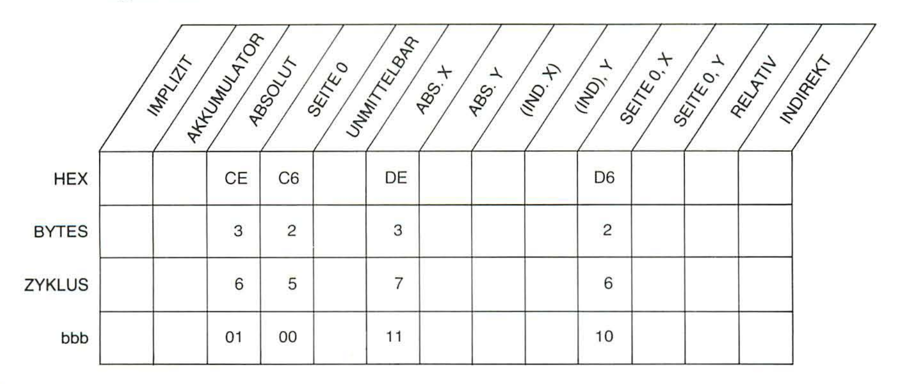
Beschreibung:
Der Inhalt der adressierten Speicherstelle wird um 1 heruntergezählt. Das Ergebnis wird an der angegebenen Adresse wieder abgelegt.



Datenwege:



Adressierungsarten:



	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			CE	C6		DE				D6			
BYTES			3	2		3				2			
ZYKLUS			6	5		7				6			
bbb			01	00		11				10			

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	11001110 bb = 01	16-Bit-Adresse HEX = CE 6 Taktzyklen
Zero-Page:	11000110 bb = 00	8-Bit-Adresse HEX = C6 5 Taktzyklen
Absolut, X:	11011110 bb = 11	16-Bit-Adresse HEX = DE 7 Taktzyklen
Zero-Page, X:	11010110 bb = 10	8-Bit-Adresse HEX = D6 6 Taktzyklen

DEX

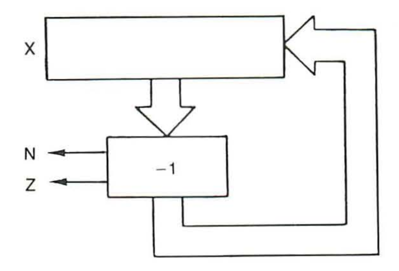
decrement X
Register X dekrementieren

Funktion: X ← (X) − 1

Format: 11001010

Beschreibung:
Der Inhalt von Register X wird um 1 heruntergezählt. Das gestattet den Einsatz von X als Zähler.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = CA, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
●					●	

DEY

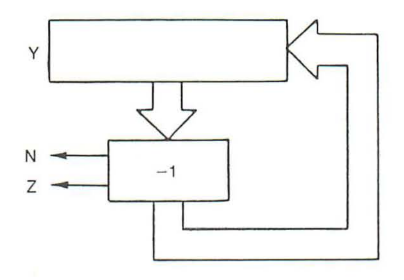
decrement Y
Register Y dekrementieren

Funktion: $Y \leftarrow (Y) - 1$

Format: 10001000

Beschreibung:
Der Inhalt von Register Y wird um 1 heruntergezählt. Das gestattet den Einsatz von Y als Zähler.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 88, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
●					●	

EOR

exclusive-OR
EXKLUSIV-ODER verknüpfen

Funktion: A ← (A) ⊻ Daten

Format:

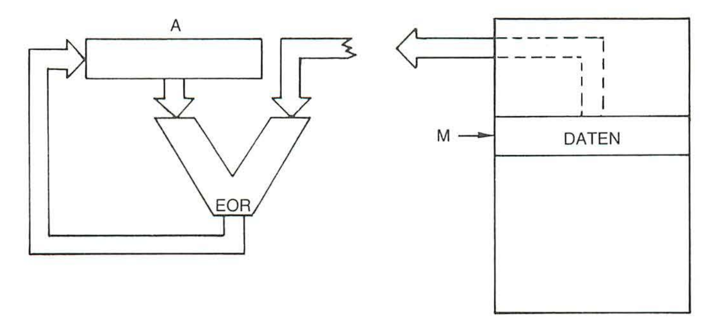
010bbb01	ADR/DATEN	ADR

Beschreibung:
EXKLUSIV-ODER verknüpft den Akkumulatorinhalt bitweise mit den angegebenen Daten. Das Ergebnis steht im Akkumulator. Die Verknüpfung erfolgt nach der Tabelle:

	0	1
0	0	1
1	1	0

Anmerkung:
EOR mit „–1“ (11111111) dient zum Komplementieren des Akkumulatorinhalts.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			4D	45	49	5D	59	41	51	55			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	01001101 bbb = 011	16-Bit-Adresse HEX = 4D 4 Taktzyklen
Zero-Page:	01000101 bb = 001	8-Bit-Adresse HEX = 45 3 Taktzyklen
Unmittelbar:	01001001 bbb = 010	Daten HEX = 49 2 Taktzyklen
Absolut, X:	01011101 bbb = 111	16-Bit-Adresse HEX = 5D 4 Taktzyklen*
Absolut, Y:	01011001 bbb = 110	16-Bit-Adresse HEX = 59 4 Taktzyklen*
(Indirekt, X):	01000001 bbb = 000	8-Bit-Adresse HEX = 41 6 Taktzyklen
(Indirekt, Y):	01010001 bbb = 100	8-Bit-Adresse HEX = 51 5 Taktzyklen*
Zero-Page, X:	01010101 bbb = 101	8-Bit-Adresse HEX = 55 4 Taktzyklen*

*: plus 1 Taktzyklus bei Seitenüberschreitung

INC

increment memory
Speicher inkrementieren

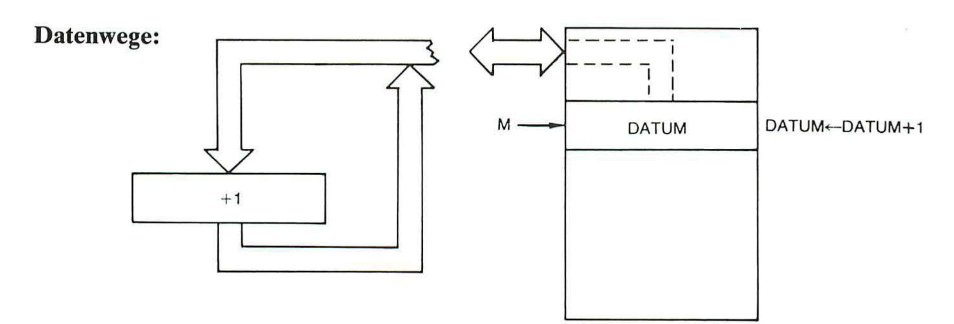
Funktion: M ← (M) + 1

Format:

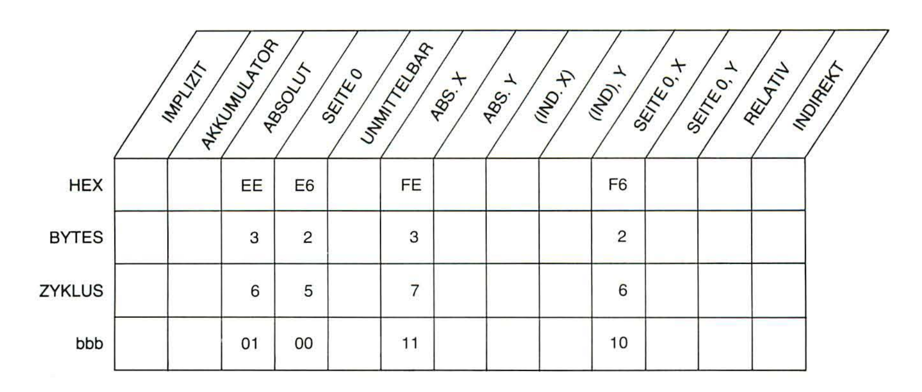
111bb110	ADR	ADR

Beschreibung:
Der Inhalt der adressierten Speicherstelle wird um 1 weitergezählt. Das Ergebnis wird an der angegebenen Adresse wieder abgelegt.

Datenwege:



Adressierungsarten:



	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			EE	E6		FE				F6			
BYTES			3	2		3				2			
ZYKLUS			6	5		7				6			
bbb			01	00		11				10			

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	11101110 bb = 01	16-Bit-Adresse HEX = EE 6 Taktzyklen
Zero-Page:	11100110 bb = 00	8-Bit-Adresse HEX = E6 5 Taktzyklen
Absolut, X:	11111110 bb = 11	16-Bit-Adresse HEX = FE 7 Taktzyklen
Zero-Page, X:	11110110 bb = 10	8-Bit-Adresse HEX = F6 6 Taktzyklen

INX

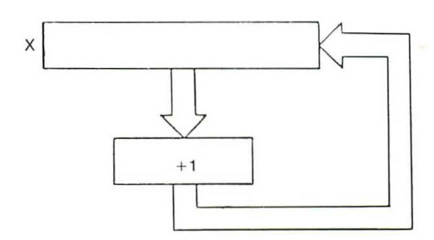
increment X
Register X inkrementieren

Funktion: X ← (X) + 1

Format: 11101000

Beschreibung:
Der Inhalt von Register X wird um 1 weitergezählt. Das gestattet den Einsatz von X als Zähler.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = E8, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
●					●	

INY

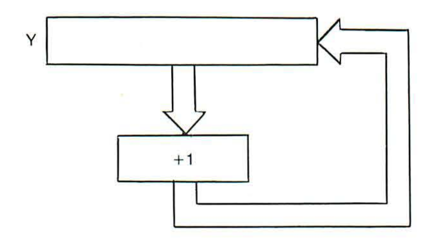
increment Y
Register Y inkrementieren

Funktion: Y ← (Y) + 1

Format: 11001000

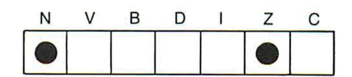
Beschreibung:
Der Inhalt von Register Y wird um 1 weitergezählt. Das gestattet den Einsatz von Y als Zähler.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = C8, 1 Byte, 2 Taktzyklen

Flaggen:



JMP

jump to address
Zur angegebenen Adresse springen

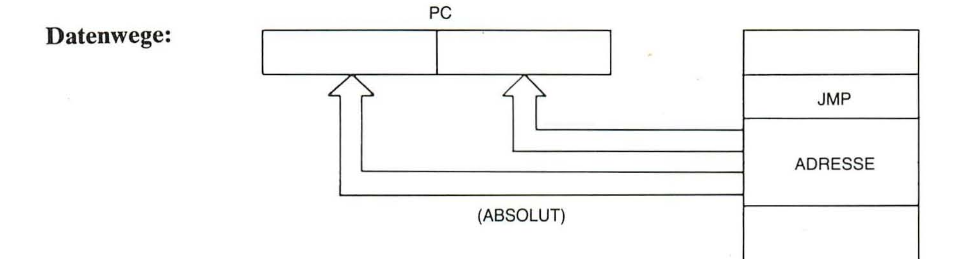
Funktion: PC ← ADR

Format:

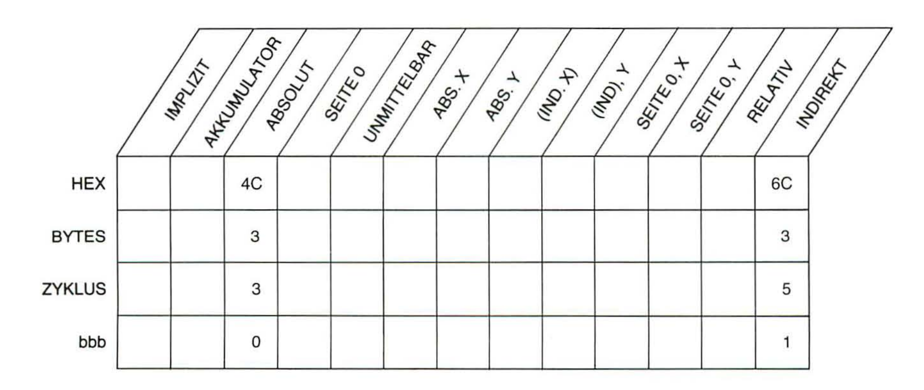
01b01100	ADR	ADR

Beschreibung:
Der Programmzähler wird mit der angegebenen Adresse geladen, was einen Sprung im Programm bewirkt. Die Adresse kann absolut oder indirekt angegeben werden.

Datenwege:



Adressierungsarten:



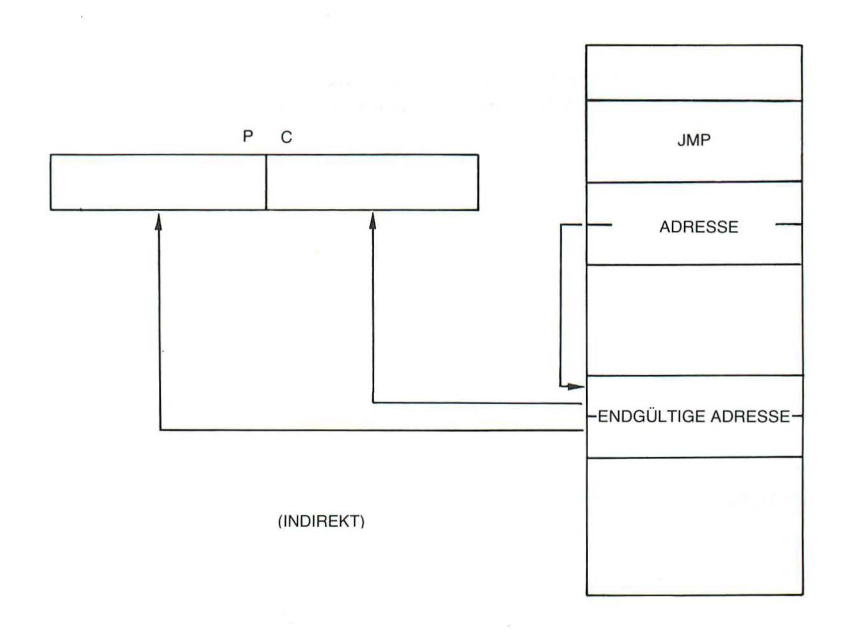
	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			4C										6C
BYTES			3										3
ZYKLUS			3										5
bbb			0										1

Flaggen:



Befehlskodes:

Absolut:	01001100 b = 0	16-Bit-Adresse HEX = 4C 3 Taktzyklen
Indirekt	01101100 b = 1	16-Bit-Adresse HEX = 6C 5 Taktzyklen



JSR

jump to subroutine
Sprung zum Unterprogramm

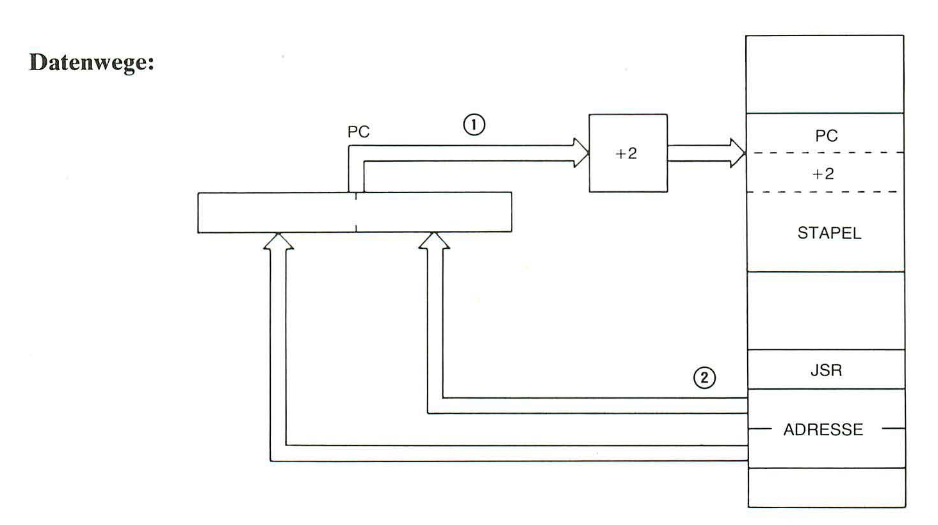
Funktion: STAPEL ← (PC) + 2
PC ← ADR

Format:

00100000	ADR	ADR

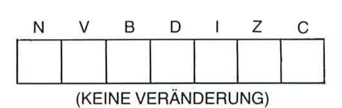
Beschreibung:
Der Programmzählerinhalt plus 2 wird auf den Stapel gebracht. (Das ist die dem JSR-Befehl folgende Adresse minus 1!) Darauf wird die Unterprogrammadresse in den Programmzähler geladen. Man bezeichnet diesen Befehl auch als Unterprogrammaufruf (subroutine CALL).

Datenwege:



Adressierungsarten:
Nur absolut:
HEX = 20, 3 Bytes, 6 Taktzyklen

Flaggen:



LDA

load accumulator
Akkumulator laden

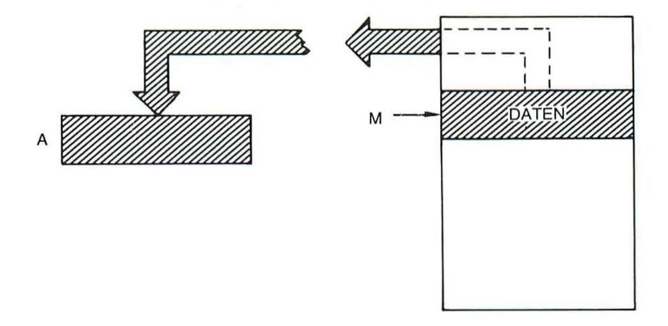
Funktion: A ← Daten

Format:

101bbb01	ADR/DATEN	ADR

Beschreibung:
Der Akkumulator wird mit neuen Daten geladen.





Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			AD	A5	A9	BD	B9	A1	B1	B5			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	10101101 bbb = 011	16-Bit-Adresse HEX = AD 4 Taktzyklen
Zero-Page:	10100101 bbb = 001	8-Bit-Adresse HEX = A5 3 Taktzyklen
Unmittelbar:	10101001 bbb = 010	Daten HEX = A9 2 Taktzyklen
Absolut, X:	10111101 bbb = 111	16-Bit-Adresse HEX = BD 4 Taktzyklen*
Absolut, Y:	10111001 bbb = 110	16-Bit-Adresse HEX = B9 4 Taktzyklen*
(Indirekt, X):	10100001 bbb = 000	8-Bit-Adresse HEX = A1 6 Taktzyklen
(Indirekt, Y):	10110001 bbb = 100	8-Bit-Adresse HEX = B1 5 Taktzyklen*
Zero-Page, X:	10110101 bbb = 101	8-Bit-Adresse HEX = B5 4 Taktzyklen

*: plus 1 Taktzyklus bei Seitenüberschreitung

LDX

load register X
Register X laden

Funktion: X ← DATEN

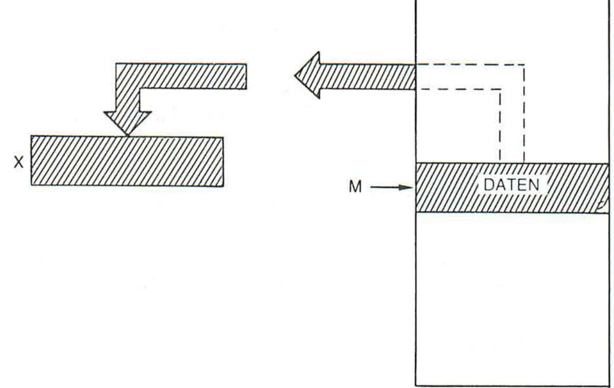
Format:

101bbb10	ADR/DATEN	ADR

Beschreibung:
Indexregister X wird mit den neuen Daten geladen.



Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			AE	A6	A2		BE				B6		
BYTES			3	2	2		3				2		
ZYKLUS			4	3	2		4*				4		
bbb			011	001	000		111				110		

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	10101110	16-Bit-Adresse
	bbb = 011	HEX = AE 4 Taktzyklen
Zero-Page:	10100110	8-Bit-Adresse
	bbb = 001	HEX = A6 3 Taktzyklen
Unmittelbar:	10100010	Daten
	bbb = 000	HEX = A2 2 Taktzyklen
Absolut, Y:	10111110	16-Bit-Adresse
	bbb = 111	HEX = BE 4 Taktzyklen*
Zero-Page, Y:	10110110	8-Bit-Adresse
	bbb = 101	HEX = B6 4 Taktzyklen

*: plus 1 Taktzyklus bei Seitenüberschreitung

LDY

load register Y
Register Y laden

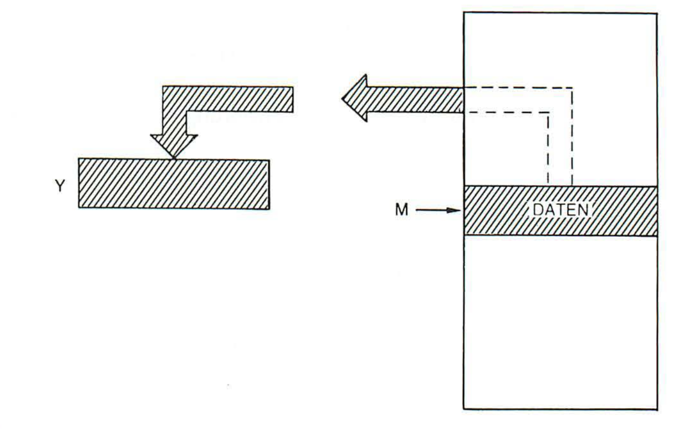
Funktion: Y ← DATEN

Format:

101bbb00	ADR/DATEN	ADR

Beschreibung:
Indexregister Y wird mit den neuen Daten geladen.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			AC	A4	A0	BC				B4			
BYTES			3	2	2	3				4			
ZYKLUS			4	3	2	4*				2			
bbb			011	001	000	111				101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	10101100 bbb = 011	16-Bit-Adresse HEX = AC 4 Taktzyklen
Zero-Page:	10100100 bbb = 001	8-Bit-Adresse HEX = A4 3 Taktzyklen
Unmittelbar:	10100000 bbb = 000	Daten HEX = A0 2 Taktzyklen
Absolut, X:	10111100 bbb = 111	16-Bit-Adresse HEX = BC 4 Taktzyklen*
Zero-Page, X:	10110100 bbb = 101	8-Bit-Adresse HEX = B4 2 Taktzyklen

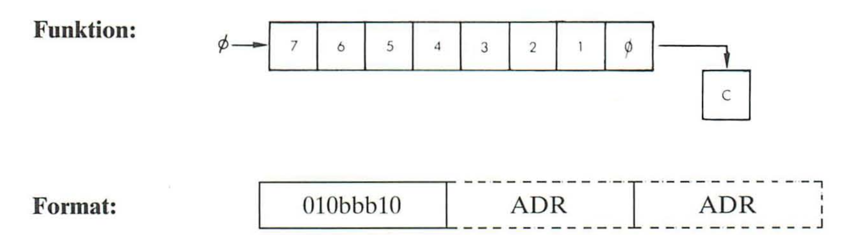
*: plus 1 Taktzyklus bei Seitenüberschreitung

LSR

logical shift right
Logisch rechts schieben

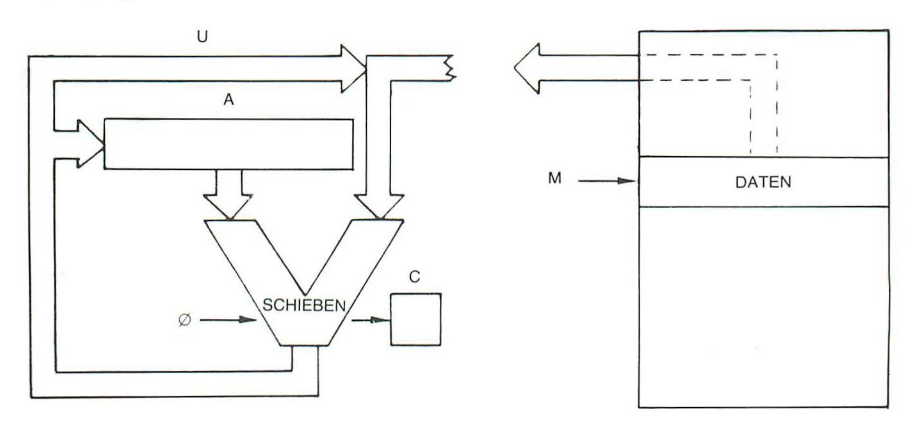
Funktion:

Format:



Beschreibung:
Akkumulator oder Speicherstelleninhalt um eine Bitstelle nach rechts schieben. Bit 7 wird zu „0“, Bit 0 wird in das Übertragsbit C geschoben. Das Ergebnis steht in der Datenquelle (Akkumulator oder Speicherstelle).

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX		1A	4E	46		5E				56			
BYTES		1	3	2		3				2			
ZYKLUS		2	6	5		7				6			
bbb		010	011	001		111				101			

Flaggen:

N	V	B	D	I	Z	C
Ø					●	●

Befehlskodes:

Akkumulator:	01001010	
	bbb = 010	HEX = 4A 2 Taktzyklen
Absolut:	01001110	16-Bit-Adresse
	bbb = 011	HEX = 4E 6 Taktzyklen
Zero-Page:	01000110	8-Bit-Adresse
	bbb = 001	HEX = 46 5 Taktzyklen
Absolut, X:	01011110	16-Bit-Adresse
	bbb = 111	HEX = 5E 7 Taktzyklen
Zero-Page, X:	01010110	8-Bit-Adresse
	bbb = 101	HEX = 56 6 Taktzyklen

NOP

no operation
Keine Operation

Funktion: Keine

Format: 11101010

Beschreibung:
Tut zwei Taktzyklen lang nichts. Man setzt diesen Befehl in erster Linie zur Verlängerung von Warteschleifen und als Platzhalter für möglicherweise einzufügende Programmbefehle ein.

Adressierungsarten:
Nur implizit:
HEX = EA, 1 Byte, 2 Taktzyklen

Flaggen:



ORA

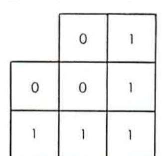
inclusive OR with accumulator
(inklusiv) mit dem Akkumulator ODER-verknüpfen

Funktion: A ← (A) V DATEN

Format:

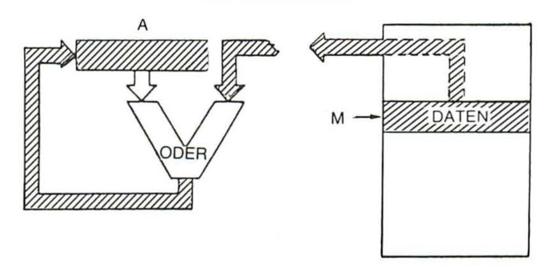
000bbb01	ADR	ADR

Beschreibung:
ODER-verknüpft den Akkumulatorinhalt bitweise mit den angegebenen Daten. Das Ergebnis steht im Akkumulator. Die Verknüpfung erfolgt nach der Tabelle:



	0	1
0	0	1
1	1	1

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			0D	05	09	1D	19	01	11	15			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●					●	

Befehlskodes:

Absolut:	00001101 bbb = 011	16-Bit-Adresse HEX = 0D 4 Taktzyklen
Zero-Page:	00000101 bbb = 001	8-Bit-Adresse HEX = 05 3 Taktzyklen
Unmittelbar:	00001001 bbb = 010	Daten HEX = 09 2 Taktzyklen
Absolut, X:	00011101 bbb = 111	16-Bit-Adresse HEX = 1D 4 Taktzyklen*
Absolut, Y:	00011001 bbb = 110	16-Bit-Adresse HEX = 19 4 Taktzyklen*
(Indirekt, X):	00000001 bbb = 000	8-Bit-Adresse HEX = 01 6 Taktzyklen
(Indirekt, Y):	00010001 bbb = 100	8-Bit-Adresse HEX = 11 5 Taktzyklen*
Zero-Page, X:	00010101 bbb = 101	8-Bit-Adresse HEX = 15 4 Taktzyklen*

*: plus 1 Taktzyklus bei Seitenüberschreitung

PHA

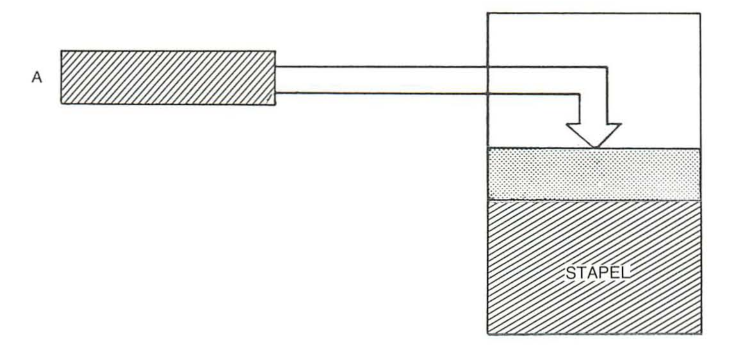
push accumulator
Akkumulator auf den Stapel bringen

Funktion: STAPEL ← (A), S ← (S) – 1

Format: 01001000

Beschreibung:
Der Akkumulatorinhalt wird auf den Stapel gebracht und der Stapelzeiger S neu gesetzt (der Stapel wächst nach unten). Der Inhalt von A bleibt unverändert.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 48, 1 Byte, 3 Taktzyklen



Flaggen:



PHP

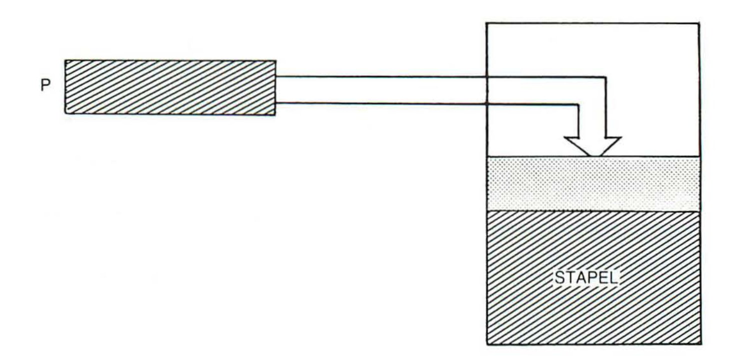
push processor status
Statusregister auf den Stapel bringen

Funktion: STAPEL ← (P), S ← (S) − 1

Format: 00001000

Beschreibung:
Der Statusregisterinhalt wird auf den Stapel gebracht und der Stapelzeiger S neu gesetzt (der Stapel wächst nach unten). Der Inhalt von P bleibt unverändert.

Datenweg:



Adressierungsarten:
Nur implizit:
HEX = 08, 1 Byte, 3 Taktzyklen

Flaggen:



PLA

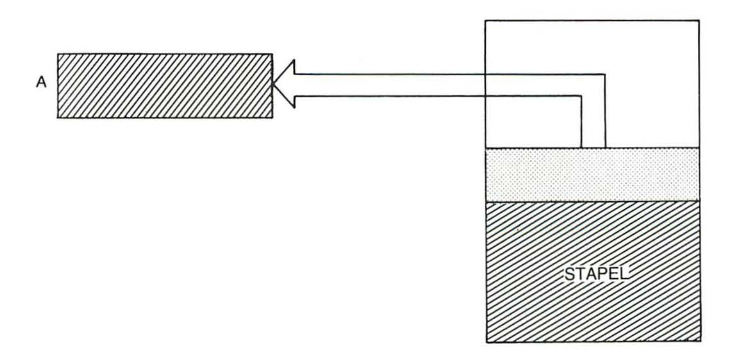
pull accumulator
Akkumulator vom Stapel holen

Funktion: A ← (STAPEL), S ← (S) + 1

Format: 01101000

Beschreibung:
Der Akkumulator wird mit dem Inhalt der Stapelspitze geladen und der Stapelzeiger S neu gesetzt (der Stapel wächst nach unten), wodurch ein Stapelelement frei wird.

Datenweg:



Adressierungsarten:
Nur implizit:
HEX = 68, 1 Byte, 4 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
●					●	

PLP

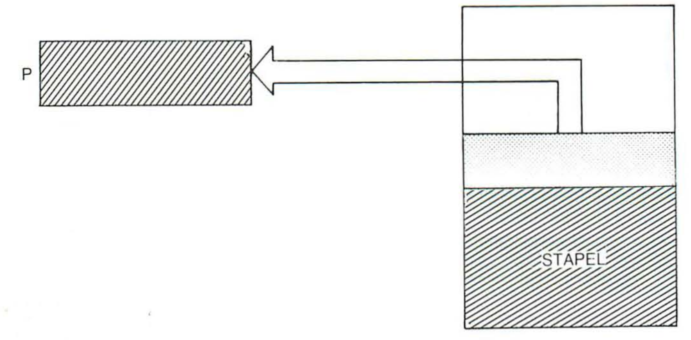
pull processor status
Statusregister vom Stapel holen

Funktion: P ← (STAPEL), S ← (S) + 1

Format: 00101000

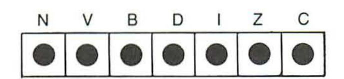
Beschreibung:
Das Statusregister wird mit dem Inhalt der Stapelspitze geladen und der Stapelzeiger S neu gesetzt (der Stapel wächst nach unten), wodurch ein Stapelelement frei wird.


Datenweg:



Adressierungsarten:
Nur implizit:
HEX = 28, 1 Byte, 4 Taktzyklen

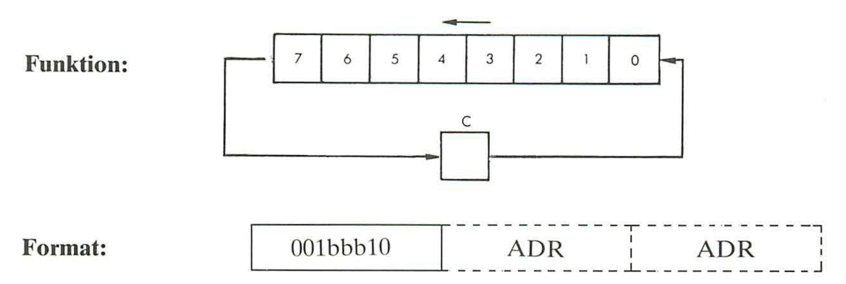
Flaggen:



ROL

rotate left one bit
Um ein Bit nach links rotieren

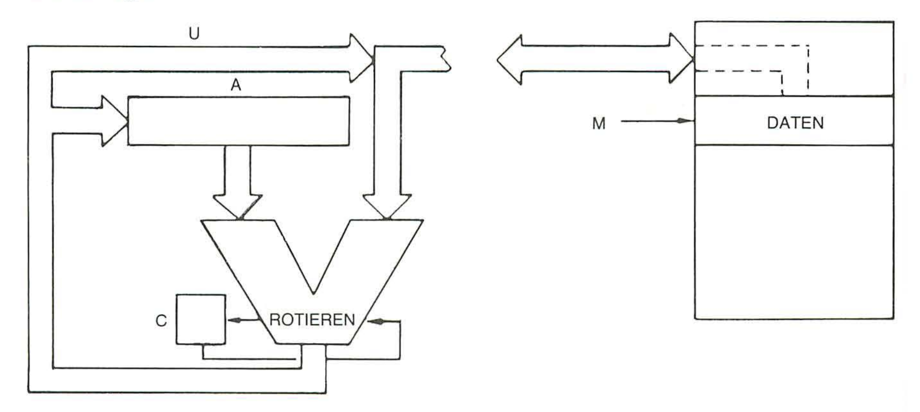
Funktion:



Format: 001bbb10 | ADR | ADR

Beschreibung:
Der Inhalt der adressierten Speicherstelle oder des Akkumulators wird um ein Bit nach links verschoben. Dabei kommt der Inhalt des Übertragsbits C in Bit 0 und Bit 7 in C, so daß der 9-Bit-Inhalt von Register und C-Bit um eine Stelle rotiert.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX		2A	2E	26		3E				36			
BYTES		1	3	2		3				2			
ZYKLUS		2	6	5		7				6			
bbb		010	011	001		111				101			

Flaggen:

N	V	B	D	I	Z	C
●					●	●

Befehlskodes:

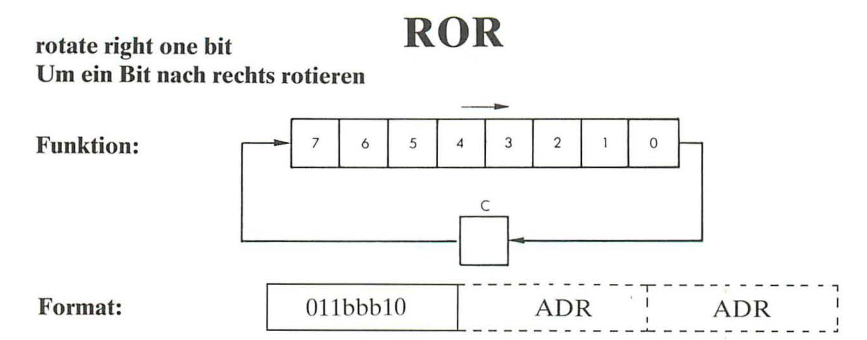
Akkumulator:	00101010 bbb = 010	HEX = 2A 2 Taktzyklen
Absolut:	00101110 bbb = 011	16-Bit-Adresse HEX = 2E 6 Taktzyklen
Zero-Page:	00100110 bbb = 001	8-Bit-Adresse HEX = 26 5 Taktzyklen
Absolut, X:	00111110 bbb = 111	16-Bit-Adresse HEX = 3E 7 Taktzyklen
Zero-Page, X:	00110110 bbb = 101	8-Bit-Adresse HEX = 36 6 Taktzyklen

ROR

rotate right one bit
Um ein Bit nach rechts rotieren

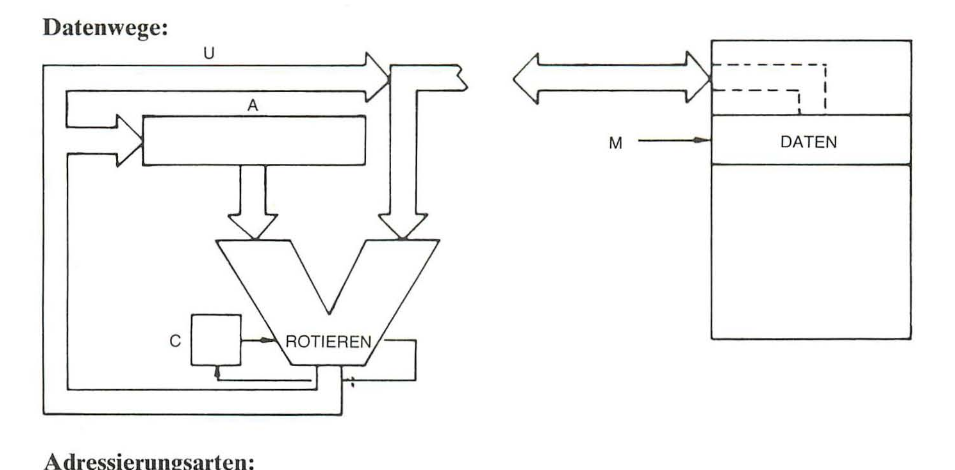
Funktion:

Format:



Beschreibung:
Der Inhalt der adressierten Speicherstelle oder des Akkumulators wird um ein Bit nach rechts verschoben. Dabei kommt der Inhalt des Übertragsbits C in Bit 7 und Bit 0 in C, so daß der 9-Bit-Inhalt von Register und C-Bit um eine Stelle rotiert.

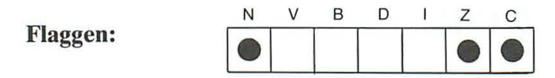
Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX		6A	6E	66		7E				76			
BYTES		1	3	2		3				2			
ZYKLUS		2	6	5		7				6			
bbb		010	011	001		111				101			

Flaggen:



Befehlskodes:

Akkumulator:	01101010 bbb = 010	HEX = 6A 2 Taktzyklen
Absolut:	01101110 bbb = 011	16-Bit-Adresse HEX = 6E 6 Taktzyklen
Zero-Page:	01100110 bbb = 001	8-Bit-Adresse HEX = 66 5 Taktzyklen
Absolut, X:	01111110 bbb = 111	16-Bit-Adresse HEX = 7E 7 Taktzyklen
Zero-Page, X:	01110110 bbb = 101	8-Bit-Adresse HEX = 76 6 Taktzyklen

RTI

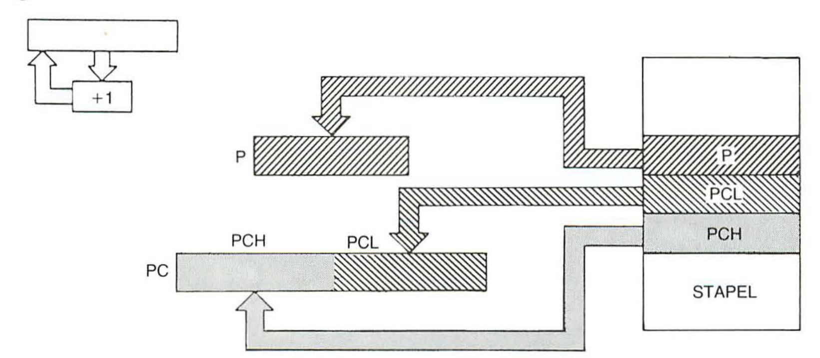
return from interrupt
Rückkehr aus einer Programmunterbrechung

Funktion:
P ← (STAPEL)
S ← (S) + 1
PCL ← (STAPEL)
S ← (S) + 1
PCH ← (STAPEL)
S ← (S) + 1

Format: 01000000

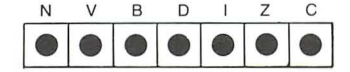
Beschreibung:
Stellt den ursprünglichen Zustand von Statusregister P und Programmzähler PC vor der Programmunterbrechung wieder her. Diese Werte wurden bei der Programmunterbrechung auf den Stapel gebracht. Der Stapelzeiger wird erneuert, womit drei Stapelplätze frei werden.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 40, 1 Byte, 6 Taktzyklen

Flaggen:



RTS

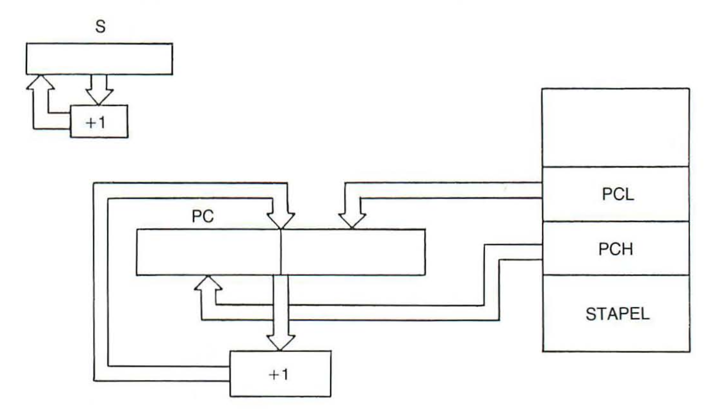
return from subroutine
Aus einem Unterprogramm zurückkehren

Funktion:
PCL ← (STAPEL)
S ← (S) + 1
PCH ← (STAPEL)
S ← (S) + 1
PC ← (PC) + 1

Format: 01100000

Beschreibung:
Programmzähler vom Stapel zurückholen und auf den nächsten Befehl stellen (d.h. um 1 weiterzählen; JSR speichert auf dem Stapel einen um 1 zu kleinen Wert!). Der Stapelzeiger wird neu gesetzt und gibt so zwei Stapeleinheiten frei.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 60, 1 Byte, 6 Taktzyklen

Flaggen:



SBC

subract with carry
Mit Übertrag subtrahieren

Funktion: A ← (A) – DATEN – C

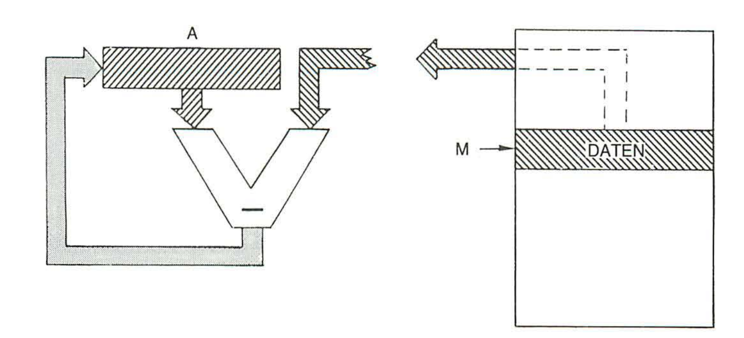
Format:

111bbb01	ADR/DATEN	ADR

Beschreibung:
Subtrahiert den Inhalt der angegebenen Speicheradresse oder Konstanten zum Akkumulator und noch den *komplementierten* Wert des Übertragsbits C. Das Ergebnis steht im Akkumulator, der (komplementierte) Übertrag in C.

Besonderheiten:
- SBC kann sowohl dezimal als auch binär arbeiten. Der Arbeitsmodus wird durch die Dezimalflagge D bestimmt, die vor der Addition auf den in Frage kommenden Wert gesetzt sein muß.
- Um ohne Übertrag zu subtrahieren, muß vor der Rechnung das Übertragsbit C mit SEC gesetzt werden.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			ED	E5	E9	FD	F9	E1	F1	F5			
BYTES			3	2	2	3	3	2	2	2			
ZYKLUS			4	3	2	4*	4*	6	5*	4			
bbb			011	001	010	111	110	000	100	101			

* Zusätzlich 1 Zyklus falls Seitengrenze überschritten wird.

Flaggen:

N	V	B	D	I	Z	C
●	●				●	●

Befehlskodes:

Absolut:	11101101 bbb = 011	16-Bit-Adresse HEX = ED 4 Taktzyklen
Zero-Page:	11100101 bbb = 001	8-Bit-Adresse HEX = E5 3 Taktzyklen
Unmittelbar:	11101001 bbb = 010	Daten HEX = E9 2 Taktzyklen
Absolut, X:	11111101 bbb = 111	16-Bit-Adresse HEX = FD 4 Taktzyklen*
Absolut, Y:	11111001 bbb = 110	16-Bit-Adresse HEX = F9 4 Taktzyklen*
(Indirekt, X):	11100001 bbb = 000	8-Bit-Adresse HEX = E1 6 Taktzyklen
(Indirekt, Y):	11110001 bbb = 100	8-Bit-Adresse HEX = F1 5 Taktzyklen*
Zero-Page, X:	11110101 bbb = 101	8-Bit-Adresse HEX = F5 4 Taktzyklen

*: plus 1 Taktzyklus bei Seitenüberschreitung

SEC

set carry
Übertragsflagge setzen

Funktion: C ← 1

Format: 00111000

Beschreibung:
Die Übertragsflagge C wird auf 1 gesetzt. Das ist oft vor einem SBC-Befehl notwendig.

Adressierungsarten:
Nur implizit:
HEX = 38, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
						1

SED

set decimal mode
Dezimalmodus einschalten

Funktion: D ← 1

Format: 11111000

Beschreibung:
Die Dezimalflagge D wird auf 1 gesetzt. Damit werden die nachfolgenden ADC- und SBC-Befehle dezimal durchgeführt, bis die Dezimalflagge wieder auf 0 gesetzt worden ist (durch CLD).

Adressierungsarten:
Nur implizit:
HEX = F8, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
			1			

SEI

set interrupt mask
Unterbrechungsmaske setzen

Funktion: I ← 1

Format: 01111000

Beschreibung:
Die Unterbrechungsmaske I im Statusregister P wird auf 1 gesetzt. Dadurch werden weitere Programmunterbrechungen gesperrt. Eine Programmunterbrechung und die allgemeine Rücksetzoperation (reset) durch Hardware setzen I immer auf 1. Um weitere Unterbrechungen zu ermöglichen, muß I ausdrücklich wieder auf 0 gesetzt werden (durch CLI).

Adressierungsarten:
Nur implizit:
HEX = 78, 1 Byte, 2 Taktzyklen

Flaggen:

N	V	B	D	I	Z	C
				1		

STA

store accumulator in memory
Akkumulatorinhalt im Speicher ablegen

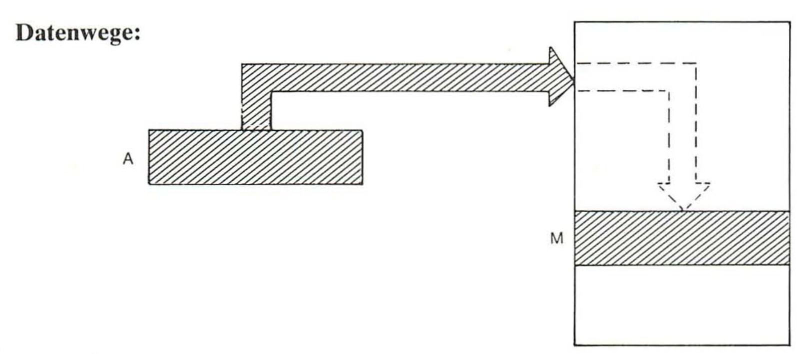
Funktion: M ← (A)

Format:

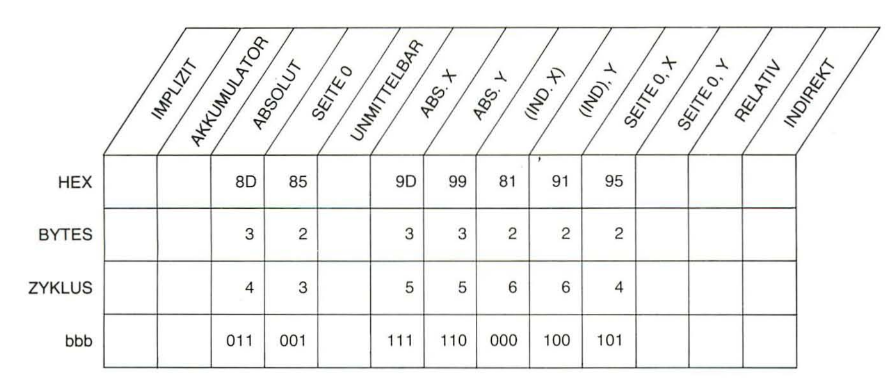
100bbb01	ADR	ADR

Beschreibung:
Der Akkumulatorinhalt wird in die adressierte Speicherstelle geschrieben. Damit bleibt der Inhalt von A unverändert.

Datenwege:



Adressierungsarten:



	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			8D	85		9D	99	81	91	95			
BYTES			3	2		3	3	2	2	2			
ZYKLUS			4	3		5	5	6	6	4			
bbb			011	001		111	110	000	100	101			

Flaggen:

N	V	B	D	I	Z	C

(KEINE VERÄNDERUNG)

Befehlskodes:

Absolut:	10001101 bbb = 011	16-Bit-Adresse HEX = 8D 4 Taktzyklen
Zero-Page:	10000101 bbb = 001	8-Bit-Adresse HEX = 85 3 Taktzyklen
Absolut, X:	10011101 bbb = 111	16-Bit-Adresse HEX = 9D 5 Taktzyklen
Absolut, Y:	10011001 bbb = 110	16-Bit-Adresse HEX = 99 5 Taktzyklen
(Indirekt, X):	10000001 bbb = 000	8-Bit-Adresse HEX = 81 6 Taktzyklen
(Indirekt, Y):	10010001 bbb = 100	8-Bit-Adresse HEX = 91 6 Taktzyklen
Zero-Page, X:	10010101 bbb = 101	8-Bit-Adresse HEX = 95 4 Taktzyklen

STX

store X in memory
Register X im Speicher ablegen

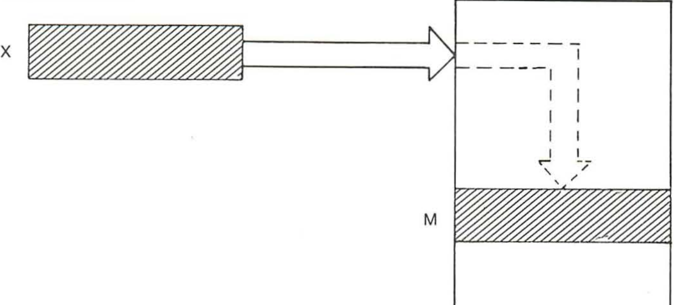
Funktion: M ← (X)

Format:

100bb110	ADR

Beschreibung:
Der Inhalt von Indexregister X wird an der adressierten Speicherstelle abgelegt. Der Inhalt von X bleibt unverändert.

Datenwege:



Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			8E	86						96			
BYTES			3	2						2			
ZYKLUS			4	3						4			
bbb			01	00						10			

Flaggen:

N	V	B	D	I	Z	C

(KEINE VERÄNDERUNG)

Befehlskodes:

Absolut:	10001110 bb = 01	16-Bit-Adresse HEX = 8E 4 Taktzyklen
Zero-Page:	10000110 bb = 00	8-Bit-Adresse HEX = 86 3 Taktzyklen
Zero-Page, X:	10010110 bb = 10	8-Bit-Adresse HEX = 96 4 Taktzyklen

STY

store Y in memory
Register Y im Speicher ablegen

Funktion: M ← (Y)

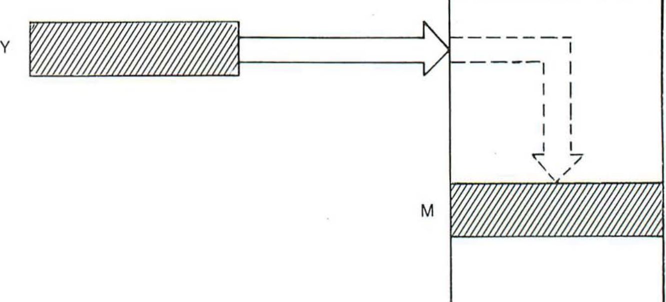
Format:

100bb100	ADR

Beschreibung:
Der Inhalt von Indexregister Y wird an der adressierten Speicherstelle abgelegt. Der Inhalt von Y bleibt unverändert.

Datenwege:





Adressierungsarten:

	IMPLIZIT	AKKUMULATOR	ABSOLUT	SEITE 0	UNMITTELBAR	ABS. X	ABS. Y	(IND. X)	(IND), Y	SEITE 0, X	SEITE 0, Y	RELATIV	INDIREKT
HEX			BC	84						94			
BYTES			3	2						4			
ZYKLUS			4	3						2			
bbb			01	00						10			

Flaggen:

N	V	B	D	I	Z	C

(KEINE VERÄNDERUNG)

Befehlskodes:

Absolut:	10001100 bb = 01	16-Bit-Adresse HEX = 8C 4 Taktzyklen
Zero-Page:	10000100 bb = 00	8-Bit-Adresse HEX = 84 3 Taktzyklen
Zero-Page, X:	10010100 bb = 10	8-Bit-Adresse HEX = 94 4 Taktzyklen

TAX

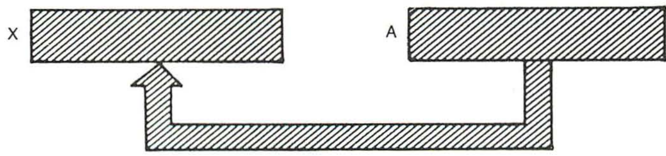
transfer accumulator into X
Akkumulator nach X übertragen

Funktion: X ← (A)

Format: 10101010

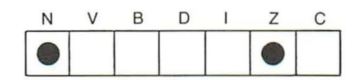
Beschreibung:
Kopiert den Akkumulator in Indexregister X. Der Inhalt von A bleibt unverändert.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = AA, 1 Byte, 2 Taktzyklen

Flaggen:



TAY

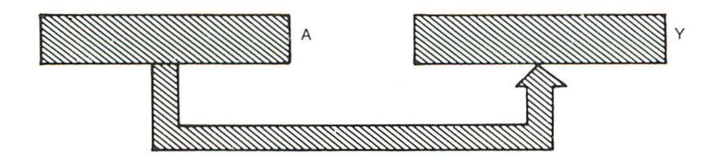
transfer accumulator into Y
Akkumulator nach Y übertragen

Funktion: Y ← (A)

Format: 10101000

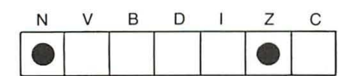
Beschreibung:
Kopiert den Akkumulator in Indexregister Y. Der Inhalt von A bleibt unverändert.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = A8, 1 Byte, 2 Taktzyklen

Flaggen:



TSX

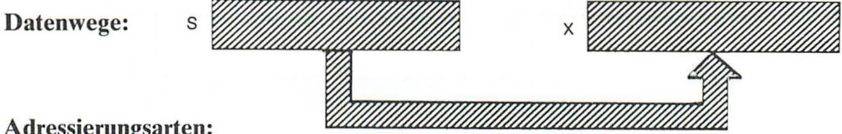
transfer S into X
Stapelzeiger S nach X übertragen

Funktion: X ← (S)

Format: 10111010

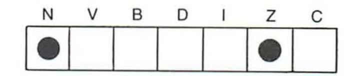
Beschreibung:
Der Inhalt des Stapelzeigers S wird in Indexregister X übertragen. Der Inhalt von S bleibt unverändert.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = BA, 1 Byte, 2 Taktzyklen

Flaggen:



TXA

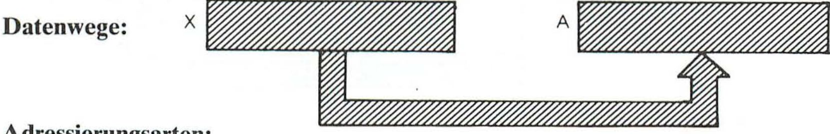
transfer X into accumulator
Register X in Akkumulator übertragen

Funktion: A ← (X)

Format: 10001010

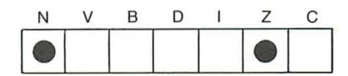
Beschreibung:
Der Inhalt von Indexregister X wird in den Akkumulator übertragen. Dabei bleibt der Inhalt von X unverändert.

Datenwege:



Adressierungsarten:
Nur implizit:
HEX = 8A, 1 Byte, 2 Taktzyklen

Flaggen:



TXS

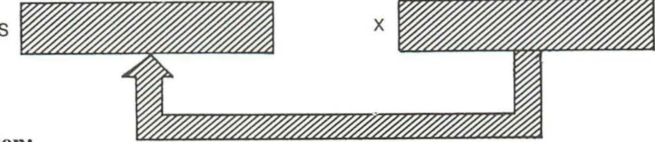
transfer X into S
Register X in den Stapelzeiger S übertragen

Funktion: S ← (X)

Format: 10011010

Beschreibung:
Der Inhalt von Indexregister X wird in den Stapelzeiger S übertragen. (Dies ist die einzige Möglichkeit, den Stapelzeiger zu laden!) Dabei bleibt der Inhalt von X unverändert.





Adressierungsarten:
Nur implizit:
HEX = 9A, 1 Byte, 2 Taktzyklen

Flaggen:



TYA

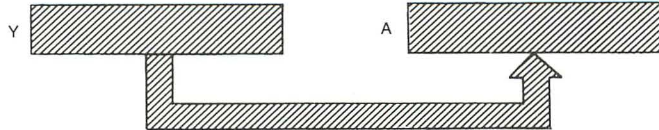
transfer Y into accumulator
Register Y in Akkumulator übertragen

Funktion: A ← (Y)

Format: 10011000

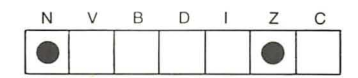
Beschreibung:
Der Inhalt von Indexregister Y wird in den Akkumulator übertragen. Dabei bleibt der Inhalt von Y unverändert.





Adressierungsarten:
Nur implizit:
HEX = 98, 1 Byte, 2 Taktzyklen

Flaggen:



KAPITEL 5
ADRESSIERUNGSARTEN

Einführung

Dieses Kapitel stellt die allgemeine Theorie der Adressierung dar, wobei die verschiedenen Techniken vorgestellt werden, die zur Wiedergewinnung von Daten entwickelt worden sind. In einem zweiten Abschnitt werden die speziell beim 6502 verfügbaren Adressierungsarten betrachtet, zusammen mit ihren etwaigen Vor- und Nachteilen. Um schließlich den Leser mit den verschiedenen Entscheidungsmöglichkeiten und Notwendigkeiten bei Anwendung dieser Techniken vertraut zu machen, wird ein besonderer Anwendungsteil derartige Abwägungen an der Praxis verdeutlichen.
Da der 6502 außer dem Programmzähler über keine 16-Bit-Register zur Adreßangabe verfügt, ist es notwendig, daß der 6502-Anwender die verschiedenen Adressierungsarten voll versteht, insbesondere den Einsatz der Indexregister. Dabei können für den Anfang komplexere Zugriffsmethoden auf die Daten, wie die Kombinationen von indirekter und indizierter Adressierung ausgelassen werden. Dennoch sind alle möglichen Adressierungsarten bei der Programmvorstellung für diesen Mikroprozessor von Bedeutung. Sehen wir uns zunächst an, was für Möglichkeiten es überhaupt gibt.

Adressierungsarten

Adressierung bezieht sich auf die Angabe in einem Befehl, wo der Operand steht, mit dem gearbeitet werden soll. Die Hauptmethoden hierfür wollen wir im folgenden untersuchen.

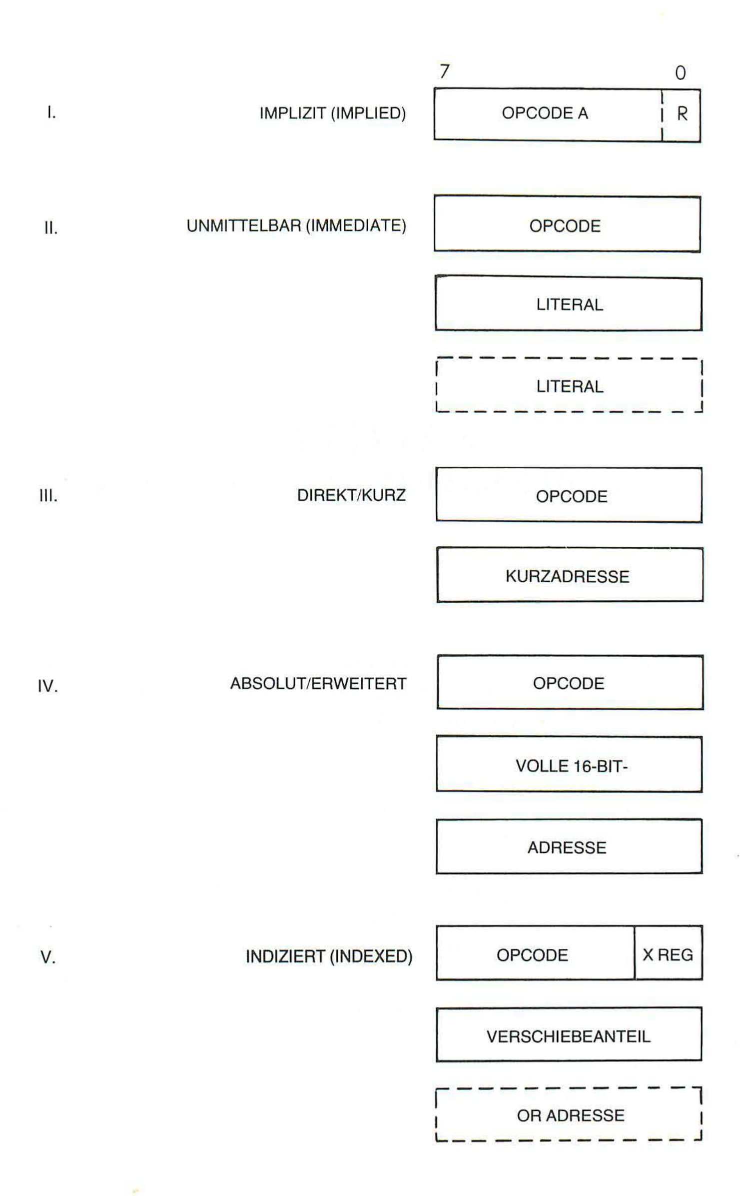


Bild 5-1: Befehlsstruktur bei den gebräuchlichen Adressierungsarten

Implizite Adressierung (implicit)

Befehle, die ausschließlich mit Registern arbeiten, benutzen normalerweise *implizite Adressierung* (implicite addressing).
Diese ist in Bild 5-1 illustriert. Ein implizit adressierender Befehl hat seinen Namen von der Tatsache, daß er die Operandenadresse nicht ausdrücklich enthält. Statt dessen gibt der Befehlskode selbst ein oder mehrere Register an, normalerweise den Akkumulator, aber auch andere Register. Da üblicherweise nur eine geringe Zahl interner Register zur Verfügung steht (sagen wir, maximal acht Stück), werden für diese Angabe nur wenige Bits benötigt. So kann man z.B. mit drei Bits in einem Befehl jedes beliebige von acht Registern bestimmen. Derartige Befehle lassen sich daher in der Regel in acht Bits unterbringen. Das ist ein wichtiger Vorzug, da ein Einbytebefehl normalerweise schneller abgearbeitet wird, als Zwei- oder Dreibytebefehle.
Ein Beispiel für einen implizit adressierenden Befehl beim 6502 ist TAX, der die Operation „Akkumulatorinhalt in Register X übertragen" bewirkt.

Unmittelbare Adressierung (immediate)

Das Format für *unmittelbare Adressierung* (immediate addressing) ist in Bild 5-1 dargestellt. Der 8-Bit-Befehlskode (Opkode) wird von einer acht oder sechzehn Bits langen Konstante („literal") gefolgt. Diese Adressierungsart benötigt man z.B. um ein 8-Bit-Register mit einem 8-Bit-Wert zu laden. Enthält der Mikroprozessor 16-Bit-Register, so kann es nötig sein, 16-Bit-Konstanten zu laden. Das hängt von der Prozessorarchitektur, seinem inneren Aufbau ab.
Ein Beispiel eines unmittelbar adressierenden Befehls wäre ADC #0. In diesem Befehl enthält das zweite Wort die Konstante „0", die zum Akkumulator addiert werden soll.

Absolute Adressierung (absolute, extended)

Absolute Adressierung bezieht sich auf die Methode, mit der üblicherweise Speicherdaten erreicht werden: Dem Befehlskode folgt eine 16-Bit-Adresse. Damit benötigt absolute Adressierung Dreibytebefehle im Programm.
Ein Beispiel für absolute Adressierung ist STA $1234. Dieser Befehl gibt an, daß der Akkumulatorinhalt in der Speicherstelle mit der hexadezimalen Adresse 1234 abgelegt werden soll.
Der Nachteil absoluter Adressierung ist die Notwendigkeit von Dreibytebefehlen. Um die Leistung des Mikroprozessors zu steigern, stellt man eine andere Adressierungsart zur Verfügung, die dasselbe mit nur einem Adreßbyte bewerkstelligt: direkte Adressierung.

Direkte Adressierung (direct, short)

In dieser Adressierungsart folgt dem Befehlskode eine Achtbitadresse, wie es in Bild 5-1 wiedergegeben ist. Der Vorteil dieses Ansatzes liegt darin, daß der Befehl nur noch zwei, statt wie bei absoluter Adressierung drei Bytes umfaßt. Der Nachteil liegt darin, daß diese Adressen nur die Werte 0 bis 255, also die Speicherstellen auf Seite

Null umfassen. Man nennt dies auch „Kurzadressierung“ (short addressing) oder „Zero-Page“-Adressierung („Nullseitenadressierung“, der englische Ausdruck „Zero-Page“ hat sich jedoch für diese Adressierungsart beim 6502 fest eingebürgert.) Wenn ein Prozessor derartige Kurzadressierung ermöglicht, so setzt man die Absolutadressierung oft als „Langadressierung“ oder „erweiterte“ Adressierung (extended addressing) dagegen ab.

Relativadressierung (relative)

Normale Sprünge erfordern für den Befehlskode acht und zur Angabe der Zieladresse weitere sechzehn Bits. Genau wie im vorigen Beispiel hat das den Nachteil von drei Befehlsbytes, d.h. von drei Speicherzugriffen zur Befehlsübernahme. Man hat mit der Relativadressierung daher eine weniger verschwenderische Methode geschaffen, die mit nur zwei Worten auskommt. Das erste ist der eigentliche Sprungbefehl, üblicherweise zusammen mit Angaben einer Testbedingung, von der der Sprung abhängig gemacht wird. Das zweite Wort gibt den Abstand (displacement) vom gegebenen Befehlsort an. Da dieser sowohl positiv (Vorwärtssprung) als auch negativ (Rückwärtssprung) sein muß, wird er zumeist als Zweierkomplementzahl angegeben, die maximale Sprungweiten von -128 (rückwärts) bzw. 127 (vorwärts) ermöglicht. Relative Sprünge oder Verzweigungen lassen sich sehr oft verwenden, da die meisten Schleifen nicht allzu lang sind. Auf diese Weise wird die Arbeitsgeschwindigkeit solcher Programmteile gegenüber 16-Bit-Adressierung oftmals beträchtlich gesteigert. (Die für den Sprung gebrauchten Zeiten addieren sich bei vielen Schleifendurchläufen auf!) Als Beispiel mag der in Kapitel 3 benutzte BCC-Befehl dienen, der bei gelöschtem Übertragsbit zu einer Programmstelle im Umkreis von 127 Bits um den Befehlsort verzweigt.

Indizierte Adressierung (indexed)

Indizierte Adressierung ist eine besonders zum schrittweisen Zugriff auf die Elemente eines Speicherblocks oder einer Tabelle geeignete Technik. Wir werden dies später in verschiedenen Beispielen verdeutlichen. Das Prinzip indizierter Adressierung liegt darin, daß ein Befehl sowohl ein Indexregister als auch eine Adresse angibt. Im allgemeinsten Fall wird der Inhalt des Indexregisters zur Adresse addiert und ergibt so die Endadresse. Man kann so z.B. als Adresse den Anfang einer im Speicher stehenden Tabelle übergeben und das Indexregister dazu benutzen, die Tabellenelemente bequem nacheinander zu erreichen. In der Praxis gibt es hierbei jedoch oft Einschränkungen, die den Umfang von Indexregister oder übergebener Adresse berühren.

Vor- und Nachindizierung (pre-indexing, post-indexing)

Man kann zwei verschiedene Formen der Indizierung unterscheiden: Vor- und Nachindizierung. Vorindizierung (pre-indexing) ist die übliche Form der indizierten Adressierung, bei der die Endadresse sich als Summe aus übergebener Adresse und Indexregisterinhalt errechnet.
Nachindizierung (post-indexing) dagegen sieht die übergebene Adreßinformation als die Adresse einer Speicherstelle an, in der erst der zum Indexregister zu addierende

Wert steht. Bild 5-2 soll das verdeutlichen. Die Endadresse ergibt sich so erst *nach* Aufsuchen der im Befehl angegebenen Speicherstelle. Im Grunde ist das eine Kombination indirekter und indizierter Adressierung. Dazu müssen wir allerdings noch die Definition indirekter Adressierung nachholen.

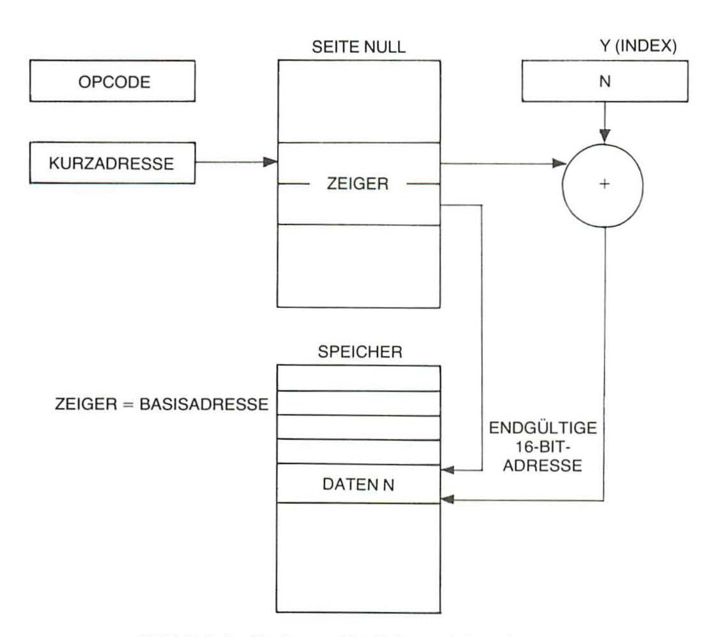


Bild 5-2: Indirekt nachindizierte Adressierung

Indirekte Adressierung (indirect)

Wir haben oben einen Fall besprochen, in dem zwei Unterprogramme eine große Datenmenge austauschen, die irgendwo im Speicher festgehalten ist. Allgemeiner ist der Fall, in dem verschiedene Programme oder Unterprogramme auf die Information in einem gemeinsamen Speicherblock zugreifen müssen. Um den Anwendungsbereich solcher Programme nicht unnötig einzuschränken, empfiehlt es sich, diese Information nicht an einer ein für alle mal festliegenden Stelle im Speicher abzulegen. Insbesondere kann die Blockgröße während der Programmabarbeitung zu- oder abnehmen, oder es kann notwendig sein, die Information bei der Abarbeitung an verschiedene Speicherstellen zu verschieben, je nachdem, wieviel Speicherplatz davor benötigt wird. Das macht es nahezu unmöglich, auf die gewünschte Information mittels absoluter Adressen zuzugreifen.
Das Problem läßt sich lösen, indem man die Anfangsadresse des Informationsblocks in einer festliegenden Speicherstelle ablegt. Das entspricht der Situation, in der mehrere Leute Zugang zu einem Haus haben müssen, zu dem aber nur ein einziger Schlüssel existiert. Man kann dann vereinbaren, den Schlüssel unter der Fußmatte zu verstecken. Jeder Hausbewohner weiß, wo er zu suchen hat (unter der Fußmatte), um

den Hausschlüssel zu bekommen. (Oder, um den Vergleich unserer Situation besser anzupassen, man findet an der vereinbarten Stelle eine Nachricht, wo ein bestimmter Treff stattfinden soll.) Indirekte Adressierung benutzt damit einen acht Bits umfassenden Befehl mit einer 16-Bit-Adreßangabe. Diese Adresse dient einfach dem Zugriff auf ein Speicherwort, das üblicherweise 16 Bits (zwei Bytes in unserem Fall) umfaßt. In Bild 5-3 ist dies verdeutlicht. Die beiden Bytes an der im Befehl angegebenen Adresse A1 enthalten eine weitere Adresse, A2. Diese Adresse A2 wird dann als die eigentlich gemeinte Ortsangabe verstanden.

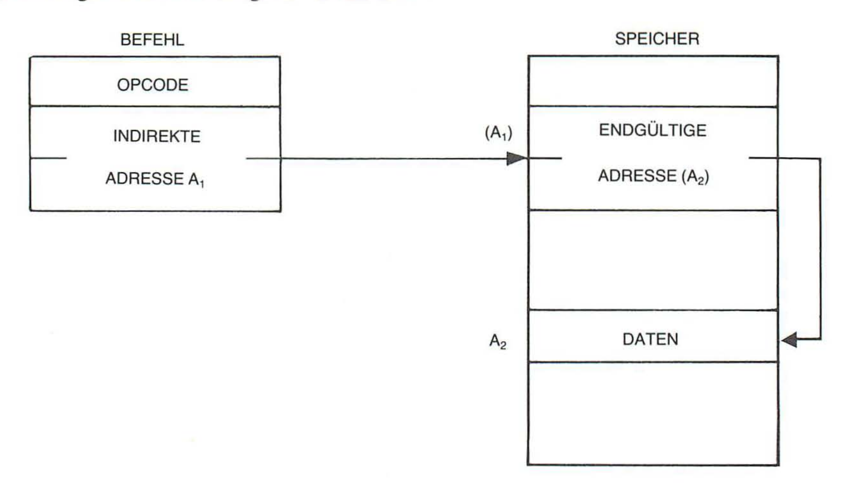


Bild 5-3: Indirekte Adressierung

Indirekte Adressierung ist insbesondere nützlich, wenn man mit Zeigern (pointers) arbeitet. In diesem Fall können die verschiedensten Programmteile sich zum Zugriff auf bestimmte Daten einfach und elegant auf diese Zeiger beziehen.

Kombination der Adressierungsarten

Die oben beschriebenen Adressierungsarten lassen sich kombinieren. Insbesondere sollte ein allgemeineres Adressierungsschema verfügbar sein, das mehrere Ebenen direkter Adressen kennt. Die Adresse A2 in Bild 5-3 würde so nicht als Endadresse, sondern als Zeiger auf eine weitere Speicherstelle, in der wieder eine Adresse steht, verstanden usw.

Ebenso läßt sich indizierte Adressierung mit indirekter verbinden. Das gestattet einen raschen und einfachen Zugriff auf das n-te Wort eines Datenblocks, dessen Anfang indirekt durch einen Zeiger im Speicher vorgegeben ist.

Damit haben wir alle für uns wichtigen Adressierungsarten kennengelernt, die ein System bieten kann. Die meisten Mikroprozessorsysteme bieten allerdings wegen der beschränkten Komplexität der MPU (die auf einem Chip untergebracht werden muß) nicht alle Möglichkeiten davon an. Der 6502 verfügt dabei über eine ungewöhnlich große Zahl an Adressierungsmöglichkeiten. Diese wollen wir uns im folgenden näher ansehen.

ADRESSIERUNGSARTEN BEIM 6502

Implizite Adressierung (6502)

Implizite Adressierung liegt bei einem Einbytebefehl vor, der mit den internen Registern arbeitet. Alle ausschließlich im 6502 ablaufenden Operationen benötigen nur 2 Taktzyklen zur Abarbeitung. Einbytebefehle, die auf den Speicher zugreifen, brauchen drei Taktzyklen.
Ausschließlich innerhalb des 6502 arbeiten: CLC, CLD, CLI, CLV, DEX, DEY, INX, INY, NOP, SEC, SED, SEI, TAX, TAY, TSX, TXA, TXS und TYA.
Speicherzugriff erfordern die Befehle: BRK, PHA, PLA, PLP, RTI und RTS.
Diese Befehle wurden im vorigen Kapitel besprochen. Die durch sie ausgeführten Operationen sollten klar sein.

Unmittelbare Adressierung (6502)

Da der 6502 nur Arbeitsregister mit 8 Bits Breite besitzt (der Programmzähler ist kein Arbeitsregister!), beschränkt sich unmittelbare Adressierung auf 8-Bit-Konstanten. Alle Befehle mit unmittelbarer Adressierung sind daher zwei Bytes lang. Das erste Byte enthält den Opkode, das zweite ergibt die in das Register zu ladende Konstante an, oder den konstanten Wert, der in die arithmetische bzw. logische Operation eingeht.
Die Befehle, die diese Adressierungsart einsetzen, lauten: ADC, AND, CMP, CPX, CPY, EOR, LDA, LDX, ORA und SBC.

Absolute Adressierung (6502)

Definitionsgemäß erfordert absolute Adressierung drei Bytes. Das erste Byte ist der Befehlskode, die beiden folgenden enthalten eine 16-Bit-Adresse (niederwertige Hälfte voran). Außer bei einem unbedingten Sprung (JMP) erfordert diese Art der Adressierung vier Taktzyklen zur Abarbeitung.
Absolute Adressierung wird bei folgenden Befehlen eingesetzt: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, JMP, JSR, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX und STY.

„Zero-Page“-Adressierung (6502)

Diese Adressierungsart erfordert definitionsgemäß zwei Bytes, eines für den Befehlskode und eines für eine 8-Bit-Kurzadresse.
„Zero-Page“-Adressierung erfordert drei Taktzyklen. Da diese Adressierungsform bedeutende Geschwindigkeitsvorteile bietet und weniger Programmkode benötigt, sollte sie wo immer möglich eingesetzt werden. Das erfordert, daß der Programmierer den benötigten Speicherplatz sorgfältig verwaltet. Allgemein lassen sich die ersten 256 Speicherstellen als Arbeitsregister des 6502 betrachten. Im wesentlichen werden alle mit diesen 256 „Registern“ arbeitende Befehle in drei Taktzyklen abgearbeitet. Daher sollte dieser Raum sorgfältig für wichtige Daten reserviert werden, auf die schnell zugegriffen werden muß.
Außer JMP und JSR können alle Befehle mit absoluter Adressierung auch mit „Zero-Page“-Adressierung arbeiten (JMP und JSR erfordern auf alle Fälle 16-Bit-Adres-

sen). Diese Befehle umfassen: ADC, AND, ASL, BIT, CMP, CPX, CPY, DEC, EOR, INC, LDA, LDX, LDY, LSR, ORA, ROL, ROR, SBC, STA, STX und STY.

Relative Adressierung (6502)

Relative Adressierung erfordert zwei Bytes. Das erste ist ein Sprungbefehl, das folgende gibt den Abstand zum Sprungziel an. Um diese Sprünge von den absolut adressierten „jumps" zu unterscheiden, nennt man sie hier „branch"-Befehle, Verzweigungen. Beim 6502 benutzen die Verzweigungen ausschließlich relative Adressierung. Normale „jump"-Befehle dagegen verwenden absolute oder indirekte Adressierung. Bei der Berechnung von Abarbeitungszeiten müssen Verzweigungsbefehle mit Vorsicht behandelt werden. Da sie immer an einen Test gebunden, also bedingte Befehle sind, bestehen mehrere Möglichkeiten. Wenn der Test negativ ausfällt, d. h. wenn nicht verzweigt wird, dann benötigt so ein Befehl zwei Taktzyklen zur Abarbeitung, denn der nächste abzuarbeitende Befehl wird in diesem Fall bereits durch den Programmzähler bezeichnet. Ist die Verzweigungsbedingung jedoch erfüllt, so werden drei Taktzyklen benötigt: die Zieladresse muß erst in den Programmzähler geladen werden. Wird dabei noch eine Seitengrenze im Speicher überschritten, so muß der Programmzählerinhalt erst korrigiert werden, so daß jetzt vier Taktzyklen notwendig sind.

Vom Standpunkt der Programmlogik aus ist es egal, ob bei einer Verzweigung eine Speicherstelle überschritten wird oder nicht. Die Mehrarbeit wird automatisch von der Hardware erledigt. Da dabei aber ein zusätzlicher Arbeitsgang und damit ein zusätzlicher Taktzyklus benötigt wird, ändert sich die Abarbeitungszeit. Man muß besonders vorsichtig sein, wenn der Verzweigungsbefehl Teil einer Verzögerungsschleife ist.

Ein guter Assembler meldet dem Programmierer einen solchen Fall, in dem die Verzweigung eine Speicherstelle überschreitet. Man kann dann Korrekturmaßnahmen vornehmen, wenn die Abarbeitungszeit kritisch ist.

Es gibt dazu noch Fälle, in denen man nicht von vornherein sagen kann, ob ein Sprung eintritt oder nicht. Das heißt, manchmal braucht der Verzweigungsbefehl zwei Taktzyklen zur Abarbeitung, manchmal drei (oder vier). Oft bestimmt man dann einen Durchschnitt, indem man beispielsweise annimmt, daß die Verzweigung in der Hälfte aller Fälle vorgenommen wird, in der anderen Hälfte dagegen nicht. Das exakte Verhältnis hängt aber von der jeweiligen Anwendung ab.

Die einzigen Befehle mit relativer Adressierung sind die acht „branch"-Verzweigungen, die die Flaggen im Statusregister als Verzweigungsbedingung ansehen: BCC, BCS, BEQ, BNE, BMI, BPL, BVC und BVS.

Indizierte Adressierung

Der 6502 bietet hier nicht den allgemeinsten Fall an, sondern macht einige Einschränkungen. Er besitzt zwei Indexregister. Diese Register sind jedoch auf acht Bits beschränkt. Der Inhalt eines dieser Indexregister wird zur im Befehl angegebenen Adresse addiert. Üblicherweise benutzt man das Indexregister als Zähler, um nacheinander auf alle Elemente eines Speicherblocks oder einer Tabelle im Speicher zugreifen zu können. Aus diesem Grund gibt es besondere Befehle, mit denen der Indexregisterinhalt inkrementiert oder dekrementiert werden kann. Zusätzlich kann man den Inhalt der Indexregister mit dem einer Speicherstelle vergleichen und so entscheiden, ob eine vorgegebene Grenze erreicht ist.

In der Praxis stört die Beschränkung der Indexregister auf acht Bits nicht so sehr, da die meisten Anwendertabellen weniger als 256 Bytes umfassen.
Die indizierte Adressierung läßt sich nicht nur zusammen mit der üblichen absoluten Adressierung benutzen, d.h. mit 16-Bit-Adressen, sondern auch bei „Zero-Page"-Adressierung, also mit 8-Bit-Kurzadressen.
Es gibt hier nur eine Einschränkung. Register X läßt sich mit beiden Formen der Bezugsadresse verwenden. Register Y jedoch gestattet nur *absolute* Angabe der Bezugsadresse und *keinen* „Zero-Page"-Modus. (Ausgenommen von dieser Regel sind die beiden Befehle LDX und STX, die sich natürlich auf Indexregister Y bei der Adreßberechnung beziehen müssen.)
Absolut indizierte Adressierung benötigt vier Taktzyklen zur Befehlsabarbeitung, solange dabei keine Seitengrenze überschritten wird, in diesem Fall werden fünf Taktzyklen notwendig.
Absolut indiziert adressierte Befehle können sowohl Register X als auch Register Y zur Indexberechnung heranziehen. Sie umfassen
- mit X: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, LSR, ORA, ROL, ROR, SBC und STA (nicht aber STY!);
- mit Y: ADC, AND, CMP, EOR, LDA, LDX, ORA, SBC und STA (nicht aber ASL, DEC, LSR, ROL, ROR und STX!)

Bei „Zero-Page"-Adressierung ist nur Indexregister X erlaubt (außer bei LDX und STX). Diese Adressierungsart ist möglich bei: ADC, AND, ASL, CMP, DEC, EOR, INC, LDA, LDY, ORA, ROL, ROR, SBC, STA und STY.

Indirekte Adressierung

Der 6502 bietet keine vollständige Möglichkeit zur indirekten Adressierung an. Er beschränkt die Adreßangabe im Befehl auf acht Bits. Mit anderen Worten: Alle indirekten Adreßangaben beziehen sich auf Seite Null im Speicher. Die vom Befehl letztlich benutzte Adresse steht dann in den beiden durch die Befehlsadresse angegebenen Speicherstellen auf Seite Null. Man kann die so gewonnene Adresse auch nicht selbst wieder als Zeiger benutzten, d.h. mehrere Ebenen indirekter Adressierung sind ausgeschlossen.
Jede indirekte Adreßangabe muß schließlich indiziert sein (außer bei JMP).
Um fair zu sein: Nur sehr wenige Mikroprozessoren bieten überhaupt indirekte Adressierung an. Des weiteren kann man ein allgemeineres indirektes Adressierungsschema durch Verwenden mehrerer Befehle nachbilden.
Es stehen (außer für den JMP-Befehl, der reine indirekte Adreßangaben benötigt) zwei Formen indirekter Adressierung zur Verfügung: indiziert-indirekte Adressierung und indirekt-indizierte Adressierung.

Indiziert-indirekte Adressierung

Hierbei wird der Inhalt von Indexregister X zur „Zero-Page"-Adresse addiert. Daraus ergibt sich ein Zeiger auf zwei Speicherstellen auf Seite Null, in denen die endgültig zu verwendende Adresse steht. Man kann so recht einfach auf verschiedene in Seite Null abgelegte Zeiger nacheinander zugreifen. Bild 5-4 verdeutlicht diese Anwendung.

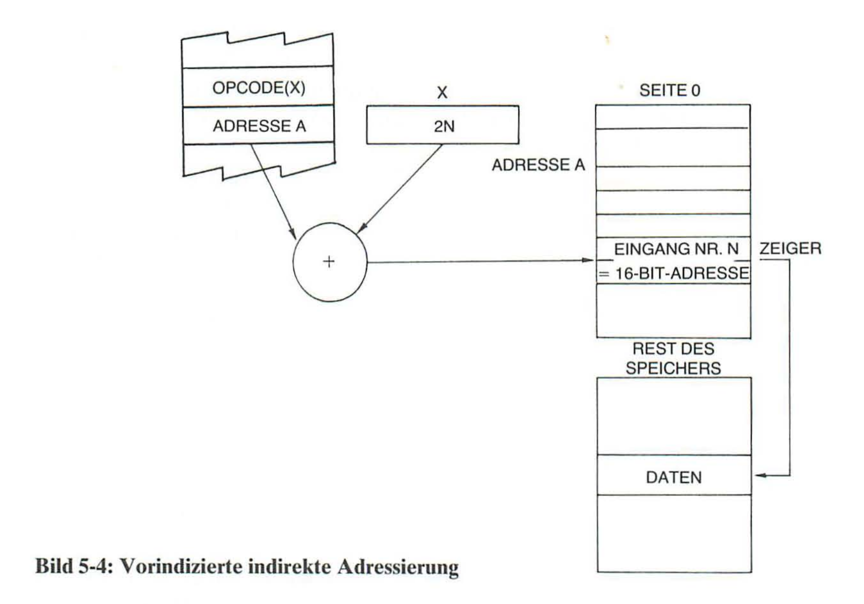


Bild 5-4: Vorindizierte indirekte Adressierung

In dieser Darstellung enthält Seite Null eine Tabelle mit verschiedenen Zeigern. Der erste davon steht unter Adresse A. Wenn nun in X der Wert 2N steht, so erhält man mit dem dazugehörigen Befehl als letztlich gültige Adresse den Wert des N-ten Zeigers aus der Tabelle.
Indiziert-indirekte Adressierung erfordert 6 Taktzyklen und ist so von der Ausführungszeit her gesehen nicht so leistungsfähig wie eine direkte Adressierungsform. Ihr Vorzug besteht in der Flexibilität beim Programmieren und in der über das Ganze möglichen Geschwindigkeitsverbesserung.
Die hiermit verwendbaren Befehle lauten: ADC, AND, CMP, EOR, LDA, ORA, SBC und STA.

Indirekt-indizierte Adressierung

Diese Form setzt den Mechanismus der Nachindizierung ein, den wir oben besprochen haben. Hier wird die Indizierung vorgenommen, nachdem die indirekt gegebene Adresse gewonnen worden ist. Mit anderen Worten: Die Kurzadresse im Befehl dient dem Zugriff auf einen Zeiger in Speicherseite Null. Zu diesem Zeiger wird dann der Index aus Register Y addiert, was die Endadresse ergibt, auf die sich der Befehl bezieht.
In diesem Fall gibt der auf Seite Null stehende Zeiger den Anfang einer Tabelle im Speicher an, und Indexregister Y enthält den Abstand von dort bis zu der gemeinten Speicherstelle. Dieser Befehl ist insbesondere nützlich bei Zugriff auf das N-te Element einer Tabelle, deren Anfangsadresse indirekt über Seite Null angegeben ist. Man benötigt dann nur zwei Bytes zum Zugriff.
Indirekt indiziert werden können die Befehle ADC, CMP, LDA, ORA, SBC und STA.

Bemerkung:

Beachten Sie die unterschiedliche Rolle der Indexregister bei indirekter Adressierung: Indexregister X dient für indiziert-indirekte, Indexregister Y für indirekt-indizierte Adressierung.

Ausnahme: Sprungbefehl JMP

Der JMP-Befehl kann nicht indiziert werden. Er verwendet unmittelbar die indirekt angegebene Adresse. Es ist der einzige Befehl, der kein Indexregister bei indirekter Adressierung einzubeziehen braucht.

EINSATZ DER ADRESSIERUNGSMÖGLICHKEITEN DES 6502

Lange Verzweigungen

Wir haben in unseren Programmen bereits „branch"-Verzweigungen verwendet. Sie bedürfen keiner weiteren Erläuterung. Interessant ist jedoch die Frage, was man tun soll, wenn der durch die relative Adressierung eines Verzweigungsbefehls mögliche Sprung zu kurz ist. Hier gibt es eine einfache Lösung, die sogenannte „lange" Verzweigung (long branch). Das ist entweder eine Verzweigung zu einem „jump"-Sprungbefehl hin oder eine Verzweigung um einen solchen Sprungbefehl herum, der dann abgearbeitet wird, wenn die Verzweigungsbedingung nicht erfüllt ist, wie im folgenden Beispiel:

```
BCC     +3      WENN C = O IST, DANN DEN FOLGENDEN
                SPRUNGBEFEHL ÜBERSPRINGEN
JMP     WEIT    SONST EINEN LANGEN SPRUNG ZU „WEIT"
                AUSFÜHREN
```

Dieses Zweizeilenprogramm verzweigt zu einer außerhalb des normalen „branch"-Bereichs liegenden Adresse WEIT, wenn C = 0 ist. Damit ist unser Problem gelöst. Sehen wir uns daher die komplexeren Adressierungsarten an, d.h. indizierte und indirekte Adreßangaben.

Adressieren aufeinanderfolgender Blockelemente

Man benutzt indizierte Adressierung in erster Linie zum Zugriff auf hintereinander in einer Tabelle stehende Werte. Dabei gilt die Einschränkung, daß die maximale Tabellenlänge 256 nicht überschreiten darf, damit der Abstand von der Tabellenspitze mit acht Bits darstellbar ist.
In Übung 3.16 haben wir gelernt, wie man nach dem ASCII-Zeichen für einen Stern (*) sucht. Wir wollen das ausbauen und eine Tabelle von 100 Elementen nach einem solchen Stern durchsuchen. Die Tabelle möge bei der Adresse BASIS beginnen und nur 100 Elemente umfassen. Damit ist sie weniger als 256 Bytes lang und kann mit Hilfe eines Indexregisters bearbeitet werden. Das Programm dafür sieht so aus:

```
SUCHEN     LDX # 0          INDEX AUF 1. ELEMENT
NAECHSTE   LDA BASIS, X     BEZEICHNETES ELEMENT HOLEN
           CMP '★'          IST ES EIN STERN?
           BEQ GEFUNDEN     JA: SCHLEIFE ABBRECHEN
           INX              SONST INDEX WEITERSETZEN
           CPX # 100        GRENZE ERREICHT?
           BNE              NAECHSTE
NICHT GEF  .......          SONST KEINEN STERN GEFUNDEN
           .......
GEFUNDEN   .......          PROGRAMMTEIL FALLS GEFUNDEN
           .......
```

Das Flußdiagramm für dieses Programm steht in Bild 5-5. Sie sollten sich von der Übereinstimmung beider überzeugen. Die Programmlogik ist dabei recht einfach: Register X dient zur Anwahl eines Tabellenelements und wird mit dem ersten Befehl auf dieses erste Element (Abstand 0 von der Tabellenspitze) gesetzt. Der zweite Programmbefehl

```
NAECHSTE LDA      BASIS, X
```

benutzt indizierte Adressierung zu einer absolut gegebenen Adresse. Er gibt an, daß der Akkumulator mit dem Inhalt der Speicherstelle geladen wird, die unter der (16-Bit-)Adresse BASIS plus dem Inhalt von Indexregister X steht. Am Anfang enthält X den Wert „0“. Damit wird als erstes auf das Element mit der Adresse BASIS zugegriffen. Im nächsten Schleifendurchlauf hat X den Wert „1“, so daß jetzt das nächste Tabellenelement unter BASIS + 1 bearbeitet wird.
Der dritte Programmbefehl, CMP '★', vergleicht das Bitmuster des übernommenen Zeichens mit dem ASCII-Wert von '★', und der nächste Befehl untersucht, was bei diesem Vergleich herausgekommen ist. Stimmen beide Bitmuster überein, so verzweigt das Programm nach GEFUNDEN:

```
         BEQ      GEFUNDEN
```

Andernfalls wird der nächste Schleifenbefehl abgearbeitet:

```
         INX
```

Das setzt den Indexzähler um 1 weiter. Aus dem Flußdiagramm in Bild 5.5 können wir entnehmen, daß jetzt der Inhalt des Indexregisters untersucht werden muß, ob wir uns noch innerhalb der Tabelle befinden. Das geschieht mit dem folgenden Befehl:

```
         CPX      #100
```

Das prüft nach, ob das Indexregister den Wert 100 erreicht hat. Falls nicht, so ist das nächste Zeichen zu untersuchen:

```
         BNE      NAECHSTE
```

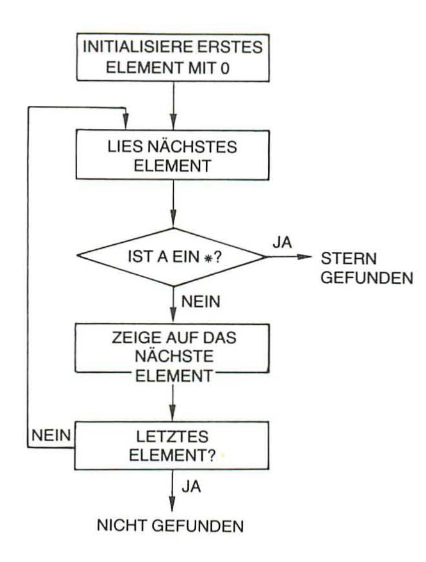


Bild 5-5: Flußdiagramm zum Durchsuchen einer Tabelle

Damit wird ein Sprung zum Label NAECHSTE (unserem zweiten Programmbefehl) durchgeführt, wenn der Test keine Gleichheit ergeben hat. Die Schleife wird daher so oft durchlaufen, bis entweder ein „*" gefunden ist oder die Tabellenlänge überschritten wurde. Im letzteren Fall ist die Suche fehlgeschlagen, und der Programmteil unter NICHTGEF ist abzuarbeiten.
Die für die Fälle „Stern gefunden" bzw. „Stern nicht gefunden" durchzuführenden Aktionen sind hier uninteressant. Sie hängen von der Aufgabe der Suche ab.
Wir haben damit gesehen, wie man indizierte Adressierung zum Zugriff auf hintereinanderstehende Tabellenelemente einsetzen kann. Diese neuerworbenen Fähigkeiten wollen wir für ein etwas schwieriges Problem benutzen. Lassen Sie uns ein wichtiges Hilfsprogramm entwickeln, mit dem man einen Speicherblock von einer Stelle zu einer anderen verschieben kann. Nehmen wir dazu zunächst an, daß der Block weniger als 256 Elemente enthält, so daß wir Indexregister X einsetzen können. Den allgemeineren Fall mit mehr als 256 Elementen wollen wir im Anschluß untersuchen.

Blockverschiebung mit weniger als 256 Elementen

Nennen wir die Elementanzahl im zu verschiebenden Block ANZAHL. Sie soll kleiner als 256 sein. BASIS sei die Anfangsadresse des Blocks, ZIEL die Anfangsadresse des Speicherbereichs, in den der Block geschoben werden soll. Der Algorithmus hierfür ist recht einfach: Wir verschieben jeweils ein Wort und vermerken seine Position in Indexregister X:

```
          LDX    # ANZAHL
NAECHSTES LDA    BASIS, X
          STA    ZIEL, X
          DEX
          BNE    NAECHSTES
```

Sehen wir uns das im Einzelnen an:

```
          LDX    # ANZAHL
```

Das lädt die Anzahl der zu übertragenden Speicherworte (Bytes) in Indexregister X. Der nächste Befehl holt dann Wort Nummer N in den Akkumulator, und der dritte überträgt diese Bytes in den entsprechenden Platz im Zielbereich. Bild 5-6 stellt dies dar.

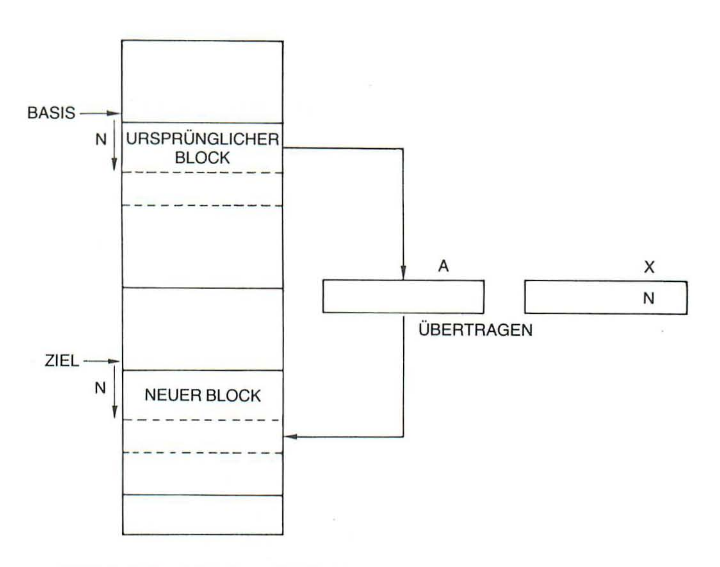


Bild 5-6: Speicherorganisation bei der Blockverschiebung

Achtung!

Dieses Programm arbeitet nur dann korrekt, wenn BASIS auf die Speicherstelle gerade *unterhalb* des zu verschiebenden Blocks zeigt. Entsprechendes gilt für ZIEL. Ist dies nicht so, muß das Programm etwas modifiziert werden.
Nach jedem übertragenen Byte muß das Indexregister angepaßt werden. Das geschieht durch den DEX-Befehl, der es jeweils um 1 herunterzählt. Das Programm testet dann nur noch, ob die Null erreicht worden ist. Falls ja, ist die Arbeit beendet. Andernfalls wird durch Rücksprung zu NAECHSTES die Schleife neu durchlaufen.

Sie werden festgestellt haben, daß bereits bei X = 0 die Schleife abgebrochen ist. Das bedeutet, daß das Wort unter BASIS nicht mehr übertragen wird. Das letzte verschobene Byte steht in BASIS + 1. Hier liegt der Grund, weshalb wir angenommen haben, daß BASIS gerade eine Stelle *unterhalb* den Block zeigt.

Übung 5.1:
Ändern Sie das Programm so ab, daß BASIS und ZIEL jeweils auf das erste Wort im Block zeigen können.

Das Programm zeigt auch den Einsatz von Schleifenzählern. Beachten Sie, daß X mit dem *Endwert* geladen, dann *heruntergezählt* und auf Null getestet worden ist. Auf den ersten Blick mag es einfacher erscheinen, mit dem Wert „0“ anzufangen und sich dann nach oben zu arbeiten, bis der Endwert erreicht ist. Das kostet aber jeweils einen eigenen Befehl zum Test, ob X seine Obergrenze erreicht hat oder nicht (einen CPX-Vergleichsbefehl). Damit würde die Schleife fünf statt vier Befehle umfassen. Nun benutzt man dieses Verschiebeprogramm oft für lange Speicherblöcke, die viele Schleifendurchgänge erfordern. Jeder zusätzlich abzuarbeitende Befehl summiert sich bei der Gesamtausführungszeit schnell auf, so daß es sehr wichtig ist, so viele Befehle wie möglich einzusparen. Aus diesem Grund zählt man – zumindest in kurzen Schleifen – das Indexregister normalerweise *herunter statt hinauf*.

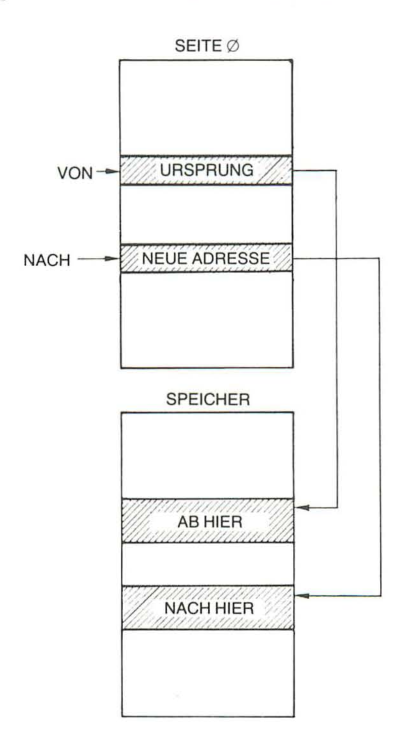


Bild 5-7: Speicher bei verallgemeinerter Blockverschiebung

Blockverschiebung mit mehr als 256 Einheiten

Betrachten wir nun den allgemeineren Fall, in dem ein Speicherblock mit mehr als 256 Elementen verschoben werden muß. Hier können wir kein einzelnes Indexregister mehr verwenden, da 8 Bits zur Speicherung einer Zahl größer als 256 nicht ausreichen. Bild 5-7 zeigt die Speicherorganisation für ein solches Programm. Wegen der Länge des zu verschiebenden Speicherblocks benötigen wir 16 Bits, die im Speicher festgehalten werden. Die höherwertige Hälfte gibt die Anzahl der 256-Byte-Blöcke an und heißt demzufolge BLOECKE. Was bleibt, wird REST genannt und stellt die Anzahl von Wörtern dar, die nach Übertragen aller Blöcke noch verschoben werden müssen. Die Adressen von Herkunft und Ziel des Blocks stehen in den Speicherstellen VON und NACH. Dann ergibt sich folgendes Programm:

```
           LDA  # QUELLNW   QUELLE, NIEDERWERTIG
           STA  VON         SPEICHERN
           LDA  # QUELLHW   QUELLE, HOEHERWERTIG
           STA  VON + 1     SPEICHERN
           LDA  # ZIELNW    ZIEL, NIEDERWERTIG
           STA  NACH        SPEICHERN
           LDA  # ZIELHW    ZIEL, HOEHERWERTIG
           STA  NACH + 1    SPEICHERN
           LDX  # BLOECKE   BLOCKANZAHL
           LDY  # 0         BLOCKLAENGE (= 256)
NAECHSTES  LDA  (VON), Y    ELEMENT HOLEN
           STA  (NACH), Y   UND UEBERTRAGEN
           DEY              ELEMENTE ZAEHLEN
           BNE  NAECHSTES   NICHT FERTIG: SCHLEIFE
NAECHSTBLK INC  VON + 1     BLOCKZEIGER, QUELLE
           INC  NACH + 1    BLOCKZEIGER, ZIEL
           DEX              BLOECKE ZAEHLEN
           BMI  FERTIG      - 1: ALLES UEBERTRAGEN
           BNE  NAECHSTES   ≠ 0: NOCH EIN BLOCK
           LDY  # REST      # 0: BLOECKE FERTIG
           BNE  NAECHSTES   REST UEBERTRAGEN, WENN ≠ 0
FERTIG     ..........
```

Die ersten vier Befehle speichern die Quellenadresse des Blocks, d. h. dessen Anfang vor der Verschiebung in den beiden Speicherstellen VON und VON + 1 ab, mit dem höherwertigen Teil in VON + 1. Die nächsten vier Befehle machen dasselbe mit der Zieladresse, die in die Speicherstellen NACH und NACH + 1 kommt. Da wir mehr als 256 Bytes verschieben müssen, benutzen wir einfach zwei Indexregister, müssen aber, da wir im folgenden indirekte Adressierung verwenden, auf die unterschiedliche Rolle von X und Y achten. So lädt der erste Befehl die Anzahl der Blöcke in Register X, das einfach als Zähler dient, während der Befehl danach Indexregister Y auf

„0“ initialisiert, um so 256-Byte-Blöcke übertragen zu können. Mit Register Y können wir indirekt-indizierte Adressierung verwenden. Erinnern Sie sich, daß indirekt-indizierte Adressierung bedeutet, daß zuerst die indirekte Adresse aus Seite Null geholt wird, und dann zu dieser Grundadresse der Index auf Register Y addiert wird, was erst die endgültige Speicheradresse für den Zugriff ergibt. Sehen wir uns das im Programm an:

```
NAECHSTES LDA (VON), Y
```

Dieser Befehl lädt den Akkumulator mit dem Inhalt (angedeutet durch die Klammern um den Namen) der Speicherstelle, deren Adresse mit dem Quellenzeiger in VON, VON + 1 plus dem Inhalt von Indexregister Y angegeben wird. Vergleichen Sie das mit der Speichereinteilung aus Bild 5-7. Im gegebenen Fall hat Y noch den Inhalt 0. In „A“ wird damit der Inhalt der Speicheradresse „QUELLE“ (source) geladen. Beachten Sie, daß im Gegensatz zu unserem vorigen Beispiel „QUELLE“ das erste Blockelement bezeichnet.

Mit derselben Technik wird im nächsten Befehl der Akkumulatorinhalt (im Augenblick das erste Wort des zu übertragenden Blocks) in der zugehörigen Zieladresse abgelegt:

```
          STA       (NACH), Y
```

Wie vorhin brauchen wir für 256 Schleifendurchgänge einfach nur das Indexregister herunterzuzählen. Das geschieht mit den nächsten beiden Befehlen:

```
          DEY
          BNE       NAECHSTES
```

Achtung!
Wir benutzen hier einen Programmiertrick, mit dem sich eine Programmverkürzung ergibt. Indexregister Y wird *dekrementiert*. Das erste übertragene Wort steht auf Stelle 0 im Block, das nächste ist Nummer 255. Dekrementieren eines auf 0 gesetzten 8-Bit-Registers ergibt den Inhalt 11111111 oder 255. Vergewissern Sie sich, daß dabei kein Fehler passieren kann. Denn wenn Register Y schließlich auf 0 heruntergezählt ist, erfolgt *keine* weitere Übertragung. Der nächste in diesem Fall ausgeführte Befehl steht unter NAECHSTBLK. D.h. in einem vollen Durchlauf der Schleife sind 256 Worte (Nummer 0 und 255 weitere) übertragen worden.
Wenn erst ein vollständiger Block übertragen worden ist, muß einfach nur in Quelle und Ziel die nächste Stelle bezeichnet werden. Das erreichen wir, indem eine „1“ zu den höherwertigen Adreßhälften addiert wird:

```
NAECHSTBLK INC      VON + 1
           INC      NACH + 1
```

Darauf müssen wir noch nachsehen, ob ein weiterer Block zu übertragen ist, indem wir den Blockzähler X herunterzählen:

```
        DEX
```

Sind alle Blöcke übertragen, so brechen wir das Programm mit einer Verzweigung nach FERTIG ab:

```
        BMI     FERTIG
```

Andernfalls blieben zwei Möglichkeiten: Entweder es sind noch Blöcke übrig (Zähler ungleich Null) oder es sind alle vollen 256-Byte-Blöcke übertragen (Zähler gleich Null). Im ersten Fall durchlaufen wir die Schleife neu mit einem Sprung zurück zu NAECHSTES:

```
        BNE     NAECHSTES
```

Sind alle vollen Blöcke übertragen, so bleibt noch der Rest. Dieser ergibt die letzte Runde in unserer Verschiebung:

```
        LDY     # REST
```

lädt den Bytezähler mit der Anzahl der noch zu bearbeitenden Bytes. Beachten Sie, daß das immer weniger als 256 Bytes sein müssen, denn sonst hätten wir ja noch einen vollen Block vor uns, der ein paar Schritte vorher erledigt würde. Andererseits kann es sein, daß kein Rest vorhanden ist, der noch zu übertragen wäre, Y also den Wert „0" erhält. Würden wir mit diesem die Schleife noch einmal durchlaufen, so würde ein Block zuviel übertragen. (Überzeugen Sie sich davon!) Mit anderen Worten: Wir dürfen nur zurückspringen, wenn Y ungleich Null ist, andernfalls sind wir mit der Arbeit fertig. Da bei jedem Laden eines Registers beim 6502 auch die Z-Flagge dem neuen Inhalt gemäß gesetzt wird, ist das hier eine einfache Sache:

```
        BNE     NÄCHSTES
```

Überzeugen Sie sich jetzt davon, daß nach diesem letzten Durchgang das Programm tatsächlich beendet wird: Wenn wir jetzt zu NAECHSTBLK kommen, wird Register X, das im letzten Durchgang auf Null heruntergezählt worden war, beim Dekrementieren negativ (11111111 = −1 im Zweierkomplement), und wir kommen automatisch nach FERTIG.

Addieren zweier Blöcke

In diesem Beispiel soll der Einsatz eines Indexregisters zur Addition zweier Datenblöcke mit weniger als 256 Einheiten illustriert werden. Das nächste Programm sieht so aus:

```
BLKADD   LDY  # ANZ - 1     ZAEHLER LADEN
NAECHSTE CLC                ADDITION VORBEREITEN
         LDA  ZGR 1, Y      NAECHSTES ELEMENT HOLEN
         ADC  ZGR 2, Y      ADDIEREN
         STA  ZGR 3, Y      UND ERGEBNIS ABLEGEN
         DEY                ALLES FERTIG?
         BPL  NAECHSTE      NEIN: SCHLEIFE
```

Indexregister Y dient als Schleifenzähler und wird mit der Anzahl der Elemente minus Eins geladen. Wir nehmen weiter an, daß ZGR 1 auf das erste Element von Block 1, ZGR 2 auf das erste Element von Block 2 und ZGR 3 auf das erste Element des Ergebnisblocks zeigt.
Das Programm erklärt sich selbst. Es wird zunächst das letzte Element von Block 1 in den Akkumulator gelesen, zu dem letzten Element des zweiten Blocks addiert und das Ergebnis als letztes Element des Ergebnisblocks abgespeichert. Dann folgt das davorstehende Element in allen Blöcken usw.

Dasselbe Beispiel mit indirekt-indizierter Adressierung

Nehmen wir an, daß die Adressen ZGR 1, ZGR 2 und ZGR 3 anfangs unbekannt sind. Allerdings wissen wir, daß sie in Seite Null unter den Adressen BEZUG 1, BEZUG 2 und BEZUG 3 stehen, was ein allgemeiner Mechanismus zur Parameterübergabe an ein Unterprogramm ist. Dann können wir das Programm so umschreiben:

```
BLKADD   LDY  # ANZ - 1      ZAEHLER LADEN
NAECHSTE CLC                 ADDITION VORBEREITEN
         LDA  (BEZUG 1), Y   NAECHSTES ELEMENT HOLEN
         ADC  (BEZUG 2), Y   ADDIEREN
         STA  (BEZUG 3), Y   UND ERGEBNIS ABLEGEN
         DEY                 ALLES FERTIG?
         BPL  NAECHSTE       NEIN: SCHLEIFE
```

Die Ähnlichkeit der beiden Programme ist offensichtlich. Hier ist die Anwendung des Mechanismus der indirekt-indizierten Adressierung deutlich, die immer dann Vorteile bringt, wenn die absoluten Adressen beim Programmieren noch nicht bekannt sind. Es sei vermerkt, daß beide Programme genau dieselbe Anzahl von Befehlen haben. Interessant wäre zu wissen, welches von beiden schneller arbeitet.

Übung 5.2:
Berechnen Sie die Byteanzahl und die Abarbeitungsdauer der beiden Programme zur Blockaddition unter Verwendung der Tabellen aus dem Anhang.

Zusammenfassung

Es wurde eine vollständige Beschreibung der üblichen Adressierungsarten gegeben. Weiter wurde gezeigt, daß der 6502 die meisten dieser Adressierungsmechanismen bietet, und ihre besonderen Eigenschaften wurden untersucht. Schließlich wurden

verschiedene Anwendungen vorgestellt, mit denen der Wert dieser Adressierungstechniken illustriert werden sollte. *Die 6502-Programmierung erfordert ein umfassendes Verständnis aller dieser Mechanismen!*

Übungen:

5.3:
Schreiben Sie ein Programm, das die ersten 10 Bytes einer Tabelle addiert, die mit der Speicherstelle BASIS anfängt. Das Ergebnis soll 16 Bits umfassen. (Es handelt sich dabei um die Berechnung einer Prüfsumme).

5.4:
Können Sie dasselbe Problem ohne Einsatz indizierter Adressierung lösen?

5.5:
Kehren Sie die Reihenfolge der 10 Bytes in dieser Tabelle um. Speichern Sie das Ergebnis unter Adresse UMKTAB ab.

5.6:
Durchsuchen Sie die Tabelle nach ihrem größten Element. Übertragen Sie dieses in die Speicherstelle GROESSTES.

5.7:
Addieren Sie die auf den einander entsprechenden Plätzen stehenden Elemente der drei Tabellen, die mit BASIS 1, BASIS 2 bzw. BASIS 3 beginnen und legen Sie das Ergebnis unter BASIS 4 ab. Die Länge der Tabellen finden Sie in Speicherstelle LAENGE.

KAPITEL 6
EIN- UND AUSGABETECHNIKEN

Einführung

Wir haben jetzt gelernt, wie man Information zwischen dem Speicher und den verschiedenen Prozessorregistern austauscht. Wir haben gelernt, wie man mit Registern umgeht und wie man die verschiedensten Befehle zur Verarbeitung der Daten einsetzt. Wir müssen noch lernen, wie man mit der Welt außerhalb des Rechners Verbindung aufnimmt. Dies nennt man Eingaben in den und Ausgaben vom Rechner.
Eingabe bedeutet die Übernahme von Daten, die außerhalb des Rechners in peripheren Einheiten (Tastatur, Diskettenstationen, Sensoren u. ä.) vorliegen. *Ausgabe* bezieht sich auf die Übertragung von Daten vom Mikroprozessor oder vom Speicher an externe Einheiten wie einen Drucker, ein Bildschirmgerät, eine Diskettenstation, an Relais und vieles mehr.
Wir werden dabei in zwei Schritten vorgehen. Zuerst werden wir lernen, wie man die von den einzelnen Einheiten geforderten Ein/Ausgabeoperationen durchführt. Dann sehen wir uns an, wie man mehrere Ein/Ausgabeeinheiten gleichzeitig bedienen kann, d. h. ihre *Verwaltung*. Der zweite Teil wird insbesondere eine Abwägung zwischen Abfragemethoden und Programmunterbrechungen bringen.

Ein- und Ausgabe

In diesem Abschnitt werden wir lernen, wie man einfache Signale wie z. B. Impulse erkennt bzw. erzeugt. Dann untersuchen wir die zur Erzeugung bzw. zum Messen exakter Zeitdifferenzen nötigen Techniken. Damit sind wir dann so weit, komplexere Ein/Ausgabearten zu betrachten, z. B. serielle und parallele Datenübertragungen bei hohen Geschwindigkeiten.

Signalerzeugung

Im einfachsten Fall muß ein externes Gerät vom Computer ein- oder ausgeschaltet werden. Um den Zustand einer Ausgabeeinheit zu ändern, ist im wesentlichen nur das Umschalten eines logischen Pegels von „0“ nach „1“ oder von „1“ nach „0“ nötig. Nehmen wir an, daß an Bit „0“ eines Registers namens AUS 1 ein Relais angeschlossen sei. Um es einzuschalten schreiben wir einfach eine „1“ in das betreffende Bit des Registers. Dabei nehmen wir hier an, daß AUS 1 in unserem System die Adresse eines Ausgaberegisters bezeichnet. Dann können wir das Relais mit dem folgenden Programm einschalten:

```
EINSCHALTEN LDA   # %00000001
            STA   AUS 1
```

Wir haben dabei vorausgesetzt, daß der Zustand der anderen 7 Bits des Registers bedeutungslos ist. Das ist jedoch oft nicht der Fall. Diese Bits können z. B. an weitere Relais angeschlossen sein. Verbessern wir daher unser einfaches Programm. Wir wollen das Relais einschalten, ohne den Zustand der anderen Bits im Register zu verändern. Nehmen wir an, daß man den Registerinhalt auch auslesen kann. Dann läßt sich das Programm so verbessern:

```
EINSCHALTEN LDA   AUS 1          INHALT DES REGISTERS HOLEN
            ORA   # %00000001    BIT 0 auf „1“ SETZEN
            STA   AUS 1          REGISTER NEU SETZEN
```

Das Programm liest zuerst den alten Stand des Ausgaberegisters AUS 1 und ODER-verknüpft diesen dann mit der Maske „00000001“. Das setzt Bit 0 auf „1“ und läßt die übrigen Bits unverändert. (Die Arbeitsweise der ODER-Verknüpfung findet sich in Kapitel 4.) Bild 6-1 verdeutlicht diesen Vorgang.

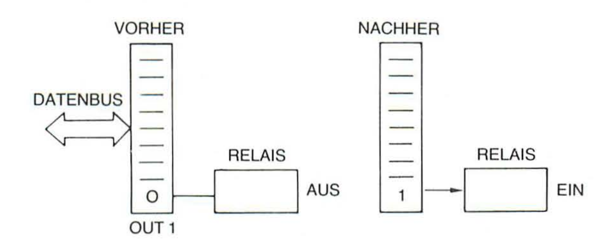


Bild 6-1: Einschalten eines Relais

Impulse

Ein *Impuls* wird ganz entsprechend dem einfachen *Spannungspegel* im vorigen Beispiel erzeugt. Zuerst wird ein Ausgabebit eingeschaltet und einige Zeit später wieder ausgeschaltet, was den gewünschten Impuls ergibt, wie Bild 6-2 zeigt. Allerdings ist diesmal ein weiteres Problem zu lösen: Der Impuls muß eine genau bestimmte Dauer haben. Untersuchen wir darum genauer, wie man eine Verzögerung erzeugt.

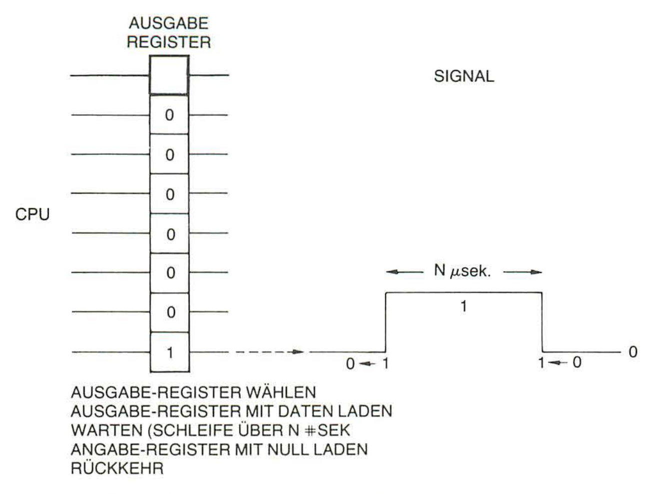


Bild 6-2: Ein programmierter Impuls

Erzeugen und Messen von Verzögerungen

Eine Verzögerung läßt sich durch Soft- und durch Hardwaremethoden erzeugen. Wir wollen hier untersuchen, wie man so etwas durch ein Programm erreicht und später zeigen, wie man einen Hardwarezähler (einen programmierbaren Intervallgeber PIT [programmable interval timer] zu diesem Zweck einsetzen kann.

Programmierte Verzögerungen erzielt man durch Zählen. Man lädt ein Zählerregister mit einem bestimmten Wert und dekrementiert es dann. Das Programm läuft in einer Schleife um, bis der Zähler die „0" erreicht hat. Die dabei benötigte Gesamtzeit ergibt die verlangte Verzögerung. Erzeugen wir als Beispiel eine Verzögerung von 37 Mikrosekunden.

```
VERZG      LDY # 07        ZEIT IN ZAEHLER Y VORBEUGEN
ZAEHLEN    DEY             HERUNTERZAEHLEN
           BNE  ZAEHLEN    BIS ZAEHLER LEER IST
```

Das Programm lädt Indexregister Y mit dem Wert 7. Der nächste Befehl zählt Y herunter und der darauffolgende Befehl verzweigt zu diesem Zählbereich zurück, solange ein Wert ungleich Null in Y steht. Das Programm verläßt die Schleife, wenn Y leergezählt worden ist und führt die dann folgenden Befehle aus. Die Programmlogik ist einfach und im Flußdiagramm in Bild 6-3 wiedergegeben.
Lassen Sie uns jetzt die Verzögerung betrachten, die sich aus dem Programm ergibt. Dazu entnehmen wir aus dem Anhang die Anzahl der für die Ausführung jedes dieser Befehle notwendigen Taktzyklen:
LDY benötigt bei unmittelbarer Adressierung (immediate) 2 Taktzyklen. DEY braucht ebenfalls 2 Zyklen zur Abarbeitung und BNE schließlich nimmt 3 Taktzyklen in Anspruch. Wenn Sie die Zyklenzahl für BNE in der Tabelle nachschlagen, so beachten Sie bitte, daß es 3 Möglichkeiten gibt: Tritt keine Verzweigung ein, braucht BNE nur 2 Taktzyklen.

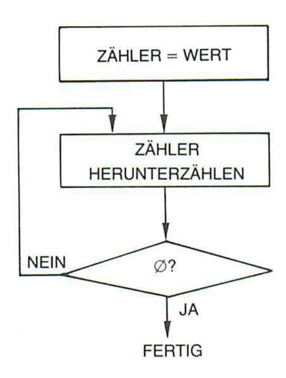


Bild 6-3: Flußdiagramm einer Verzögerungsschleife

Wird, wie es beim Schleifendurchlauf der Normalfall ist, verzweigt, so ist ein Zyklus mehr notwendig. Und wenn dabei eine Speicherseite überschritten wird, braucht man noch einen Taktzyklus mehr. Wir wollen hier annehmen, daß keine Seitengrenze überschritten wird.
Damit ergibt sich folgende Rechnung: 2 Taktzyklen für den ersten Befehl plus 5 Taktzyklen für die beiden anderen, multipliziert mit der Zahl der Schleifendurchgänge, in unserem Beispiel also mal 7:

Verzögerung = 2 + 5 x 7 = 37 Taktzyklen

Nehmen wir eine Zyklusdauer von einer Mikrosekunde an, so erhalten wir durch das Programm eine Verzögerung von 37 Mikrosekunden.
Da kein 6502-Befehl weniger als 2 Taktzyklen umfaßt, beträgt die kleinstmögliche Verzögerung 2 Mikrosekunden. Das bedeutet auch, daß wir ein Verzögerungsprogramm mit einer kleinsten Auflösung von zwei Mikrosekunden justieren können.

Übung 6.1:
Wie groß ist die mit diesen drei Befehlen erreichbare maximale Verzögerungszeit?

Übung 6.2:
Modifizieren Sie das Programm für eine Verzögerung von 100 Mikrosekunden.

Wenn die Verzögerung länger sein soll, so fügt man am einfachsten zwischen DEY und BNE weitere Befehle ein. Der bequemste Weg ist das Verlängern mit Hilfe von NOP-Befehlen. (Ein NOP-Befehl macht 2 Taktzyklen lang gar nichts!)

Längere Verzögerungen

Längere Verzögerungen durch Software erhält man, wenn man einen längeren Zähler einsetzt. Zwei interne Register, besser noch zwei Speicherworte lassen sich als 16-Bit-Zähler verwenden. Nehmen wir der Einfachheit halber an, das niederwertige By-

te habe den Inhalt „0“. Dann wird es beim ersten Durchgang durch die Zählschleife auf „225“ gesetzt und von da an kontinuierlich pro Schleifendurchgang um 1 heruntergezählt. Wenn „0“ erreicht ist, zählt man das höherwertige Zählerbyte um „1“ herunter und leitet eine neue Runde für den niederwertigen Zähler ein. Das Programm wird beendet, wenn der höherwertige Zähler auf „0“ heruntergezählt worden ist. Die Präzision läßt sich steigern, wenn der niederwertige Zählerteil einen Wert ungleich Null enthält. In diesem Fall können wir das Programm wie beschrieben aufstellen und am Ende das Dreibyteprogramm von oben anfügen.
Noch längere Verzögerungen erhält man, wenn man mehr als zwei Worte für den Zähler verwendet. Das entspricht ganz dem Funktionsprinzip des Kilometerzählers im Auto: Wenn das rechts von einem anderen stehende Rad von „9“ nach „0“ geht, so wird das links davon stehende um „1“ weitergezählt. Es ist das allgemeine Zählprinzip in Stellenwertsystemen.
Der Haupteinwand gegen diese Methode ist allerdings die Tatsache, daß der Mikroprozessor bei längeren Verzögerungen nichts anderes macht als hunderte von Mikrosekunden oder gar Sekunden zu zählen und nichts als zu zählen. Wenn der Computer sonst nichts zu tun hat, geht das ganz in Ordnung. In der Regel jedoch sollte der Mikrocomputer für andere Aufgaben bereitstehen, so daß längere Verzögerungen normalerweise nicht mittels Software durchgeführt werden. Mehr noch läßt sich in einem System, das in bestimmten Situationen rasch reagieren muß, sogar der Wert von kurzen Zählschleifen bezweifeln. Hier muß man Hardwareverzögerungen einsetzen. Wenn zusätzlich noch Programmunterbrechungen durch Hardware (interrupts) verwendet werden sollen, geht die Verzögerungsgenauigkeit unter Umständen bei Eintritt einer Unterbrechung ganz verloren.

Übung 6.3:
Schreiben Sie ein Programm, das (etwa für einen Fernschreiber) eine Verzögerung von 100 ms erzeugt.

Verzögerungen durch Hardware

Hardwareverzögerungen werden durch einen programmierbaren Verzögerungsbaustein, kurz einen „Zeitgeber“ (timer) erzeugt. Man lädt ein Register im Zeitgeber mit einem bestimmten Wert. Der Unterschied besteht darin, daß diesmal der Zeitgeber automatisch diesen Zähler herunterzählt, wobei die Geschwindigkeit üblicherweise vom Programmierer einstell- oder vorwählbar ist. Wenn der Zeitgeber dann nach „0“ heruntergezählt ist, fordert er in der Regel vom Prozessor eine Programmunterbrechung an. Er kann aber auch ein bestimmtes Statusbit setzen, das dann regelmäßig vom Computer abgefragt werden muß. Der Einsatz von Programmunterbrechungen soll allerdings erst weiter hinten im Kapitel beschrieben werden.
Andere Zeitgeberfunktionen können das Messen der Dauer eines Signals umfassen, indem von „0“ an aufwärts gezählt wird, oder auch einfach nur das Zählen bestimmter von außen eintreffender Impulse. Einige Zeitgebereinheiten können sogar mehrere Register enthalten und eine Anzahl zusätzlicher vom Programm auswählbarer Möglichkeiten bieten. Das ist z. B. bei dem Zeitgeber im 6522-Baustein der Fall, einer Ein/Ausgabeeinheit, die wir im nächsten Kapitel betrachten werden.

Impulse messen

Das Problem, die Länge von Impulsen zu bestimmten, ist gerade das Gegenteil der Impulserzeugung, mit einer zusätzlichen Schwierigkeit: Während ein auszugebender Impuls programmgesteuert erzeugt werden kann, treten Eingabeimpulse *asynchron* zum Programm auf. Um die Anwesenheit eines Impulses zu erkennen, gibt es zwei Möglichkeiten: *Abfrage* (polling) und *Programmunterbrechung* (interrupt). Unterbrechungen werden weiter hinten im Kapitel besprochen.
Betrachten wir die Abfragetechnik. Hierbei liest das Programm regelmäßig den Inhalt eines bestimmten Registers aus, testet ein vereinbartes Bit, sagen wir: Bit 0. Nehmen wir weiter an, Bit 0 habe ursprünglich den Wert „0“. Wenn ein Impuls eintrifft, werde das Bit auf „1“ gesetzt. Das Programm überwacht nun Bit 0 dieses Registers so oft es geht, bis es dort den Wert „1“ entdeckt. Ist dies der Fall, so wurde der Anfang eines Impulses gefunden. Im einfachsten Fall läßt sich das durch folgende Schleife erledigen:

```
ABFRAGE  LDA  # $01     TESTMASKE LADEN
TESTEN   BIT  EINGANG   IST IN „EINGANG“ BIT 0 = 1?
         BEQ  TESTEN    NEIN: NEU TESTEN
EIN      ..........
         ..........
```

Das Abfrageregister EINGANG wird durch den BIT-Befehl mit der Maske 00000001 UND-verknüpft. Ist Bit 0 gleich „0“, so beträgt das Ergebnis auch Null, und die Z-Flagge im Statusregister P wird auf „1“ gesetzt. Andernfalls ist das Ergebnis ungleich Null, Z = 0, und die Schleife kann verlassen werden.
Ganz entsprechend sieht der Fall aus, in dem die Eingangsleistung normalerweise den Wert „1“ hat und bei Eintreffen des Impulses auf „0“ gesetzt wird. Das ist zum Beispiel der Fall bei Fernschreibverbindungen, wenn man den Anfang eines gesendeten Bytes, das „Starbit“ erkennen muß. Wir brauchen hier nur den Verzweigungsbefehl zu ändern:

```
ABFRAGE  LDA  # $01     TESTMASKE LADEN
TESTEN   BIT  EINGANG   IST IN „EINGANG“ BIT 0 = 0?
         BNE  TESTEN    NEIN: NEU TESTEN
START    ..........
         ..........
```

Messen der Impulsdauer

Die Dauer eines Impulses läßt sich im Prinzip genauso messen, wie man einen Impuls erzeugt: Man zählt die Länge auf. Dazu läßt sich entweder eine Hard- oder Softwaretechnik einsetzen. Beim Messen der Impulslänge durch Software wird ein Zähler jeweils um 1 weitergesetzt und dann nachgeschaut, ob der Impuls noch anliegt. Ist das der Fall, so wird zum Zählen zurückgesprungen. Ist der Impuls dagegen beendet, so kann man den Zählerstand zur Berechnung der Impulsdauer benutzen. Das zugehörige Programm sieht so aus:

```
DAUER    LDX  # 0       ZAEHLER LOESCHEN
         LDA  # $01     MASKE FUER BIT 0 SETZEN
TESTEN   BIT  EINGANG   IMPULS VORHANDEN?
         BEQ  TESTEN    NEIN: EINGANG AUF ABFRAGEN
MESSEN   INX            IMPULSDAUER ZAEHLEN
         BIT  EINGANG   IMPULS NOCH VORHANDEN?
         BNE  MESSEN    JA: WEITERZAEHLEN
FERTIG   ..........
         ..........
```

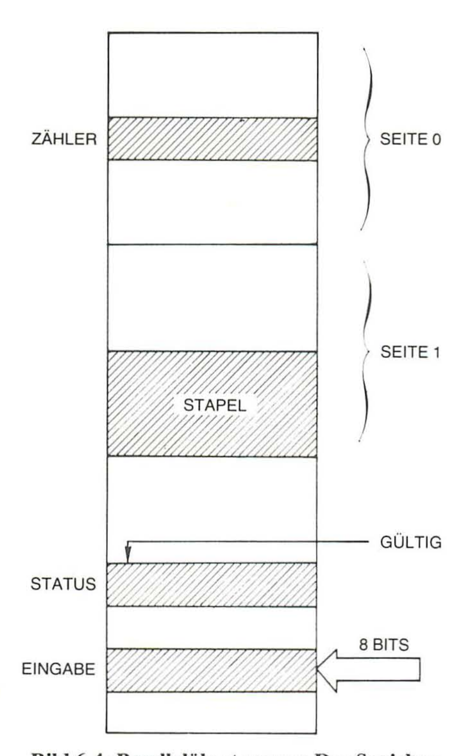


Bild 6-4: Parallelübertragung: Der Speicher

Wir nehmen dabei natürlich an, daß der Impuls nicht so lang ist, daß Register X überläuft. Wenn das eintreten kann, muß entweder die Meßschleife länger sein (bei Verlust an Genauigkeit), oder man muß weitere Bytes für den Zähler vorsehen. Andernfalls fängt X einfach wieder vorne an zu zählen, und wir bekommen den falschen Wert für die Impulslänge heraus (haben also einen Programmierfehler gemacht!).
Da wir nun wissen, wie man Impulse erzeugt und wie man sie mißt, wollen wir uns einem anderen Problem zuwenden: der Übernahme bzw. dem Senden größerer Datenmengen. Dabei sollen zwei Fälle unterschieden werden: serielle und parallele Datenübertragung. Die hier gewonnenen Kenntnisse wollen wir dann mit realen Ein/Ausgabeeinheiten anwenden.

Parallele Datenübertragung

Nehmen wir an, daß unter der Adresse „EINGABE" 8 Bits übertragene Daten bereitstehen. Der Mikroprozessor muß das an dieser Stelle anliegende Datenwort immer dann übernehmen, wenn ein bestimmtes Statuswort anzeigt, daß die Daten gültig sind. Nehmen wir also weiter an, daß diese Statusinformation in Bit 7 unter der Adresse „STATUS" übergeben wird. Damit können wir hier ein Programm erstellen, das die ankommenden Daten alle liest und automatisch abspeichert. Um die Sache zu vereinfachen, nehmen wir noch an, daß die Zahl der übertragenen Datenworte unter der Adresse „ANZAHL" im Voraus angegeben wird. Wäre diese Information nicht vorhanden, so müßten wir den Datenstrom nach einem *Endzeichen* untersuchen, z.B. einem Löschzeichen (rubout, ASCII 7F hexadezimal) oder einem Stern („★", ASCII 2D). Wie man das macht, wissen wir bereits.

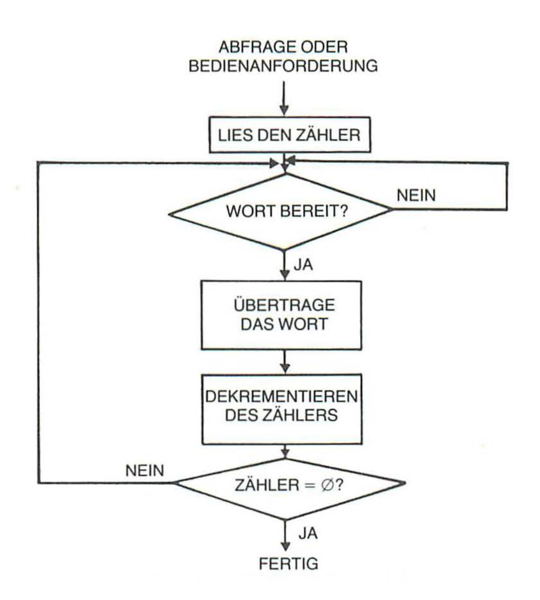


Bild 6-5: Parallelübertragung: Flußdiagramm

Für den Fall, daß die Anzahl vorgegeben ist, können wir das Flußdiagramm aus Bild 6-5 verwenden. Es ist nicht weiter kompliziert. Wir testen die Statusinformationen, bis eine „1" vorliegt, was anzeigt, daß ein gültiges Wort vorhanden ist. Dieses lesen wir dann und speichern es an der passenden Stelle ab. Dann wird der Zähler um 1 dekrementiert und getestet, ob alle Bytes übernommen worden sind (Zählerstand = 0). Falls ja, sind wir fertig. Andernfalls müssen wir das nächste Wort übernehmen. Das diesem Algorithmus entsprechende Programm folgt:

```
PARALLEL  LDX  ANZAHL    ZAEHLER LADEN
WARTEN    LDA  STATUS    IST EIN WORT VORHANDEN?
          BPL  WARTEN    NEIN: WEITERWARTEN
          LDA  EINGABE   SONST WORT UEBERNEHMEN
          PHA            UND IM STAPEL ABLEGEN
          DEX            ALLES UEBERNOMMEN?
          BNE  WARTEN    NEIN: NAECHSTES WORT HOLEN
FERTIG    ..........
          ..........
```

Mit dem ersten Befehl übernehmen wir die Anzahl der zu übertragenden Bits in Indexregister X, das uns hier wieder als Zähler dient. Die beiden folgenden Befehle dienen zum Abfragen des Statusbits, ob ein gültiges Wort vorliegt. Diese Schleife wird so lange durchlaufen, bis Bit 7 des externen Statusregisters auf „0“ steht. (Es handelt sich um das Vorzeichenbit, das beim Laden automatisch in die N-Flagge übertragen wird.)

```
WARTEN    LDA  STATUS
          BPL  WARTEN
```

Fällt der BPL-Test negativ aus, dann liegen gültige Daten vor, die wir im folgenden übernehmen können:

```
          LDA  EINGABE
```

Damit ist das Byte von dem Register unter Adresse EINGABE gelesen und muß noch gespeichert werden. Unter der Annahme, daß die Anzahl der zu übertragenden Bytes klein genug ist, wollen wir dazu den Stapel verwenden:

```
          PHA
```

Wenn der Stapel voll ist, oder wenn sehr viele Datenworte übertragen werden, können wir den Stapel nicht zu diesem Zweck verwenden. Wir müssen dann einen speziell dafür bereitgestellten Speicherbereich nehmen, wozu wir z.B. indizierte Adressierung einsetzen können. Dies würde jedoch einen besonderen Befehl zum Weiterzählen des Indexregisters erfordern. PHA ist da schneller.
Damit ist das Datenwort übernommen und abgespeichert. Dekrementieren wir einfach noch den Zähler und sehen nach, ob wir mit der Arbeit fertig sind:

```
          DEX
          BNE  WARTEN
```

Wir bleiben in der Übernahmeschleife, solange der Zähler noch nicht bis auf Null herunter ist.
Man kann dieses 6-Befehls-Programm als sogenanntes *Benchmark-Programm* verwenden. Ein Brenchmark-Programm (wörtlich Programm zum Beurteilen auf dem Arbeitstisch [bench]) ist ein sorgfältig optimiertes Programm für einen gegebenen

Prozessor in einer bestimmten Situation. Parallele Datenübertragung ist eine solche, oft auftretende Situation. Das Programm wurde auf maximale Arbeitsgeschwindigkeit und Platzausnutzung hin entworfen. Errechnen wir nun seine maximale Übernahmegeschwindigkeit. Dabei sei angenommen, daß ANZAHL in Seite Null steht. Die Abarbeitungsdauer jedes Befehls ergibt sich aus der Tabelle im Anhang wie folgt:

```
                                  Taktzyklen
PARALLEL  LDX  ANZAHL             3
WARTEN    LDA  STATUS             4
          BPL  WARTEN             3/2 (verzw./nicht verzw.)
          LDA  EINGABE            4
          PHA                     3
          DEX                     2
          BNE  WARTEN             3/2 (verzw./nicht verzw.)
FERTIG    ..........
          ..........
```

Die minimale Ausführungszeit ergibt sich, wenn wir annehmen, daß ein Wort immer dann anliegt, wenn wir STATUS abfragen. Anders herum gesagt soll jeder BPL-Test auf Anhieb negativ ausfallen. Dann erhalten wir bei einer Zykluszeit von einer Mikrosekunde als kürzeste Übernahmezeit:

$3 + (4 + 2 + 4 + 3 + 2 + 3) \times$ ANZAHL Mikrosekunden

Vernachlässigen wir die ersten drei Mikrosekunden zur Initialisierung des Zählerregisters, so erhalten wir für ein Wort einen Zeitbedarf von 18 Mikrosekunden. Daraus ergibt sich eine maximale Übertragungsrate von

$$\frac{1}{18 \times 10^{-6}} = 55 \text{ Kbytes pro Sekunde}$$

Übung 6.4:
Nehmen Sie an, die Anzahl der übertragenen Worte sei größer als 256. Ändern Sie das Programm entsprechend ab und berechnen Sie die Auswirkungen auf die Übertragungsrate.

Damit haben wir gelernt, wie man sehr schnelle parallele Datenübertragungen ausführen kann. Sehen wir uns im folgenden einen etwas komplexeren Fall an.

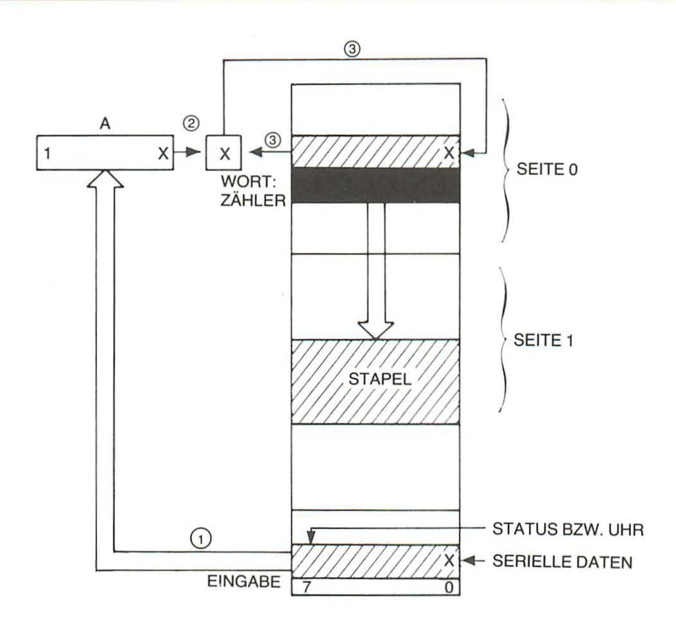


Bild 6.6: Seriell/Parallel-Umwandlung: Register- und Speicherbelegung

Serielle Datenübertragung

Bei einem seriellen Dateneingang kommen die Informationsbits („0" oder „1") nacheinander über eine einzige Leitung an. Diese Bits können in regelmäßigen Abständen eintreffen, was man üblicherweise als *synchrone* Übertragung bezeichnet. Oder sie kommen zu nicht vorhersehbaren Zeiten in ganzen Gruppen kurz nacheinander an. In diesem Fall spricht man von *asynchroner* Datenübertragung. Wir wollen ein Programm entwickeln, das beide Fälle handhaben kann. Das Grundprinzip zur Übernahme serieller Daten ist einfach: Wir überwachen eine Eingabeleitung, die an Bit 0 eines Eingabetors angeschlossen sein möge. Wenn auf dieser Leitung ein Datenbit vorgefunden wird, so lesen wir es ein und schieben es in ein Register, das als Zwischenspeicher dient. Immer wenn so acht Bits zusammengefaßt sind, speichern wir das Datenbyte ab und übernehmen das nächste. Zur Vereinfachung nehmen wir wieder an, die Anzahl der zu übernehmenden Bytes sei im Vorhinein bekannt. Andernfalls müßten wir nach einem besonderen Endzeichen suchen und bei seinem Auftreten die serielle Datenübertragung abbrechen. Wie man das macht, haben wir bereits gelernt. Das Flußdiagramm für ein solches Programm findet sich in Bild 6-7, und das Programm selbst hat folgende Form:

```
SERIELL  LDA  # $00    UEBERNAHMEREGISTER
         STA  WORT     LOESCHEN
WARTEN   LDA  EINGABE  BIT VORHANDEN? (STATUSBIT = BIT 7)
         BPL  WARTEN   NEIN: NEU TESTEN
         LSR  A        SONST BIT VON STELLE 0 IN C
         ROL  WORT     UND VON DA AN IN REGISTER SCHIEBEN
         BCC  WARTEN   NICHT VOLL: SCHLEIFE
         LDA  WORT     SONST DAS DATENWORT HOLEN
         PHA           UND AUF DEN STAPEL BRINGEN
         LDA  # $01    UEBERNAHMEREGISTER
         STA  WORT
         DEC  ANZAHL   ALLE BYTES UEBERNOMMEN?
         BNE  WARTEN   NEIN: NAECHSTES BYTE HOLEN
FERTIG   ..........
         ..........
```

Dieses Programm ist recht leistungsfähig und benutzt einige neue Techniken, die wir im folgenden ansehen wollen (vgl. Bild 6-6).
Wir haben dabei folgende Vereinbarungen getroffen: Die Anzahl der zu übernehmenden Bytes soll in der Speicherstelle ANZAHL stehen. Speicherstelle WORT dient als Übernahmeregister, in dem die einzeln eintreffenden Bytes zu 8-Bit-Worten zusammengefaßt werden. Unter Adresse EINGABE befindet sich ein Eingaberegister, bei dem auf Bit 7 ein Status- oder Taktsignal liegen möge. Wenn es den Wert „0“ hat, sind keine gültigen Daten vorhanden, hat es den Wert „1“, so liegt an Bit 0 von EINGABE ein zu übernehmendes Datenbit an. In vielen anderen Fällen steht die Statusinformation in einem anderen Register als die Dateninformation. Es ist jedoch recht einfach, das Programm einer solchen Situation anzupassen. Wir wollen weiter annehmen, daß das erste zu übernehmende Datenbit garantiert eine „1“ ist, die dem eigentlichen Datenstrom vorangeht. Für den Fall, daß das nicht vorgesehen ist, werden wir später eine passende Änderung vorfinden. Das Programm entspricht genau dem Flußdiagramm aus Bild 6-7. Zunächst wird das Übernahmeregister auf Null initialisiert. Das ist notwendig, um später die Übernahme eines Datenworts feststellen zu können. Die nächsten beiden Zeilen stellen eine Warteschleife dar, ob ein Datenbit bereitsteht. Dazu lesen wir das Eingaberegister in den Akkumulator und testen die dabei gesetzte N-Flagge (= Bit 7 des Registers). Solange dieses Bit auf „0“ steht, ist die Verzweigungsbedingung im BPL-Bereich erfüllt, und das Programm springt zum Anfang der Warteschleife zurück. Wenn Status- oder Taktbit auf „1“ gesetzt sind und so anzeigen, daß Daten zu übernehmen sind, dann fällt der BPL-Test negativ aus, und der nächste Programmbefehl kommt an die Reihe.
Erinnern Sie sich, daß BPL „branch on plus“ bedeutet, d.h. dann verzweigt, wenn das Vorzeichenbit 7 des Registers auf „0“ steht. Dieser Eingangsabschnitt des Programms entspricht Pfeil 1 in Bild 6-6.

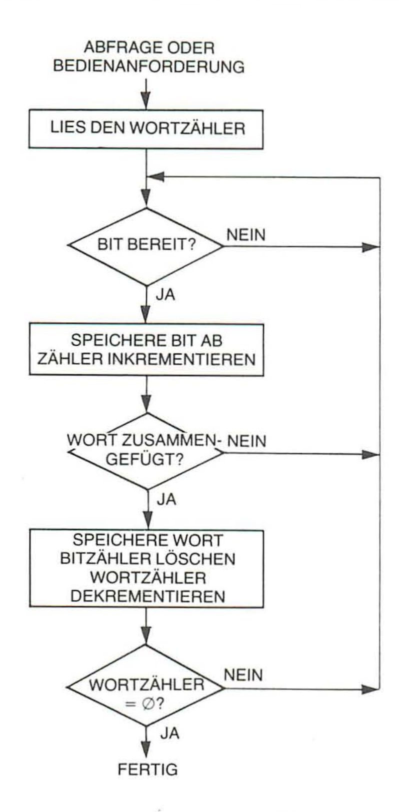


Bild 6-7: Bitserielle Übertragung: Flußdiagramm

An dieser Stelle enthält der Akkumulator in Bit 7 eine „1" und in Bit „0" das gerade übergebene Datenbit. Das erste eintreffende Bit soll nach unserer Vereinbarung eine „1" sein, die folgenden können je nach Information beliebig Einsen oder Nullen darstellen. Wir müssen jetzt das übernommene Datenbit aus Stelle „0" im Akkumulator in den Speicher übertragen. Der Befehl:

```
LSR  A
```

schiebt den Akkumulatorinhalt um eine Stelle nach rechts, was unser ganz rechts stehendes Datenbit in das Übertragsbit C bringt. Dieses Datenbit soll dann im Übernahmeregister WORT gesammelt werden (das entspricht Pfeil 2 und Pfeil 3 in Bild 6-6):

```
ROL  WORT
```

Mit diesem Befehl wird das Übertragsbit in die äußerst rechte Bitstelle von WORT geschoben. Zugleich gelangt das ganz links stehende Bit von WORT in den Übertrag C. (Wenn Sie wegen der Arbeitsweise des ROL-Befehls unsicher sind, schlagen Sie in Kapitel 4 nach).

Es ist wichtig, sich zu merken, daß ein Rotierbefehl sowohl den alten Wert des C-Bits im bearbeitenden Register festhält (hier ganz rechts) als auch den Übertrag mit dem auf der anderen Seite des Registers stehenden Bit (hier Bit 7) neu definiert.
An dieser Stelle hier wird C auf „0" gesetzt. Der nächste Befehl:

```
BCC   WARTEN
```

untersucht diesen Übertragswert und verzweigt zurück zur Warteschleife, wenn hier eine „0" vorgefunden wird. Damit haben wir einen automatischen Bitzähler. Denn im Ergebnis des ersten ROL WORT haben wir im Übernahmeregister WORT den Inhalt „00000001" stehen. Acht Rotierbefehle später wird C schließlich auf „1" gesetzt und damit der Rücksprung zur Schleife unterbrochen. Wir können so einen Schleifenzähler aufbauen, ohne Zählbefehle auf einem Indexregister verschwenden zu müssen. Diese Technik verhilft hier zur Programmverkürzung und Verbesserung seiner Leistungsfähigkeit.
Wenn der BCC-Test schließlich negativ ausfällt, haben wir in WORT acht Datenbits nacheinander übernommen. Dieser Wert muß in den Speicher gerettet werden, denn WORT brauchen wir für die nächsten Übernahmeoperationen. Die nächsten beiden Befehle dienen zu diesem Zweck (Pfeil 4 in Bild 6-6):

```
LDA   WORT
PHA
```

Wir speichern damit den Inhalt von WORT (8 Bits) auf dem Stapel ab. Das ist allerdings nur dann möglich, wenn dort noch genügend Platz ist. Unter der Voraussetzung, daß diese Bedingung erfüllt ist, bietet der Stapelspeicher den schnellsten Weg, Daten im Speicher aufzuheben. Der Stapelzeiger wird automatisch neu gesetzt. Wenn wir das Wort nicht auf dem Stapel ablegen könnten, wäre ein weiterer Befehl zur Anpassung eines Speicherzeigers notwendig. Wir könnten genausogut eine indiziert adressierte Operation durchführen, doch das würde ebenso In- oder Dekrementieren des Indexregisters und mithin mehr Zeit erfordern.
Nachdem das erste Datenwort gespeichert worden ist, gibt es keine weitere Garantie dafür, daß das nächste Bit eine „1" ist. Es kann beides sein, „0" oder „1". Wir müssen daher den Inhalt von WORT auf „00000001" zurücksetzen, wollen wir ihn weiterhin als Bitzähler verwenden. Das geschieht mit den nächsten beiden Befehlen:

```
LDA   # $01
STA   WORT
```

Schließlich haben wir den Bytezähler zu dekrementieren, denn ein Byte ist übernommen worden, und wir müssen testen, ob wir mit der Arbeit fertig sind. Die beiden letzten Programmbefehle sind hierfür da:

```
DEC   ANZAHL
BNE   WARTEN
```

Das oben wiedergegebene Programm ist auf größtmögliche Geschwindigkeit hin entworfen worden, damit man rasch hintereinander eintreffende Datenbits übernehmen kann. Wenn das Programm dann mit der Arbeit fertig ist, empfiehlt es sich, sofort die auf dem Stapel zwischengespeicherten Daten von dort herunterzuholen und woanders im Speicher unterzubringen. Wir haben derartige Blockverschiebungen in Kapitel 5 kennengelernt.

Übung 6.5:
Berechnen Sie die Maximalgeschwindigkeit mit der dieses Programm serielle Bits entgegennehmen kann. Nehmen Sie zur Geschwindigkeitsberechnung an, daß WORT und ANZAHL in Seite Null stehen. Weiter soll das gesamte Programm auf einer einzigen Speicherseite untergebracht sein. Schlagen Sie die Anzahl der von jedem Befehl benötigten Taktzyklen im Anhang nach und berechnen Sie daraus die Zeit, die zum Abarbeiten des Programms benötigt wird. Zur Berechnung der von einer Schleife erforderten Abarbeitungszeit multiplizieren Sie einfach die Zeit für einen Durchlauf mit der Zahl der Durchläufe. Des weiteren sollten Sie zur Berechnung der maximalen Übernahmegeschwindigkeit annehmen, daß bei jedem Test von EINGABE bereits wieder ein Bit bereitsteht.

Dieses Programm ist schwieriger zu verstehen als die vorangehenden. Sehen wir es uns noch einmal mehr im Detail an (Bild 6-6) und untersuchen wir seine Eigenschaften.
Von Zeit zu Zeit kommt an Bit 0 von EINGABE ein Datenbit an. Es möge sich dabei beispielsweise um drei Einsen hintereinander handeln. Wir müssen diese nacheinander eintreffenden Bits irgendwie voneinander unterscheiden können. Das ist Aufgabe eines Taktsignals.
Das Taktsignal (oder Statussignal) gibt an, daß der am Biteingang anliegende Spannungspegel für ein gültiges Datenbit steht.
Daher testen wir vor der Übernahme eines Datenbits erst das Statusbit. Ist der Status auf „0“, so müssen wir warten. Ist er aber auf „1“, so können wir das Bit übernehmen. Wir haben hier angenommen, daß dieses Statussignal über Bit 7 des Registers EINGABE zu erreichen ist.

Übung 6.6:
Können Sie erklären, warum wir Bit 7 für den Status und Bit 0 für die Daten genommen haben?

Wenn wir erst einmal ein Datenbit übernommen haben, müssen wir es irgendwo im System sicher festhalten und dann das nächste Bit übernehmen. Gleichzeitig wollen wir die einkommenden Bits zu 8-Bit-Worten zusammenfassen, so daß wir die einzeln übernommenen Bits am besten zunächst in ein Sammelregister einschieben.
Leider ist das einzige Prozessorregister, dessen Inhalt verschoben werden kann, der auch zum Test des Statusbits herangezogene Akkumulator. Würden wir ihn zum Sammeln der Eingabe benutzen, so würde bei jedem neuen Test von EINGABE das eben übernommene Bit verlorengehen.

Übung 6.7:
Können Sie eine Möglichkeit angeben, das Statusbit zu testen, ohne dabei den Akkumulatorinhalt zu löschen (durch Verwenden eines besonderen Befehls)? Wenn das möglich ist, können wir dann den Akkumulator benutzen, um die ankommenden Bits zu einem Byte zusammenzufassen?

Übung 6.8:
Schreiben Sie das Programm so um, daß der Akkumulator die übernommenen Bits zusammenfaßt. Vergleichen Sie es zu dem obenstehenden Hinblick auf Geschwindigkeit und Anzahl der Befehle.

Sehen wir uns zwei andere mögliche Variationen an:
Wir haben angenommen, daß das allererste Bit in unserem Beispiel ein besonderes Signal, d.h. mit Sicherheit eine „1" sein muß. Im allgemeinen Fall jedoch kann es jeder beliebige Wert sein.

Übung 6.9:
Ändern Sie das Beispielprogramm unter der Annahme ab, daß das erste übernommene Bit ein gültiges Datenbit ist, das nicht verlorengehen darf und sowohl eine „1" als auch eine „0" sein kann. Hinweis: Unser „Bitzähler" muß trotzdem richtig arbeiten. Sie müssen ihn mit dem richtigen Wert initialisieren.

Schließlich haben wir noch das zusammengefaßte Byte aus dem Arbeitsregister von WORT in den Stapel geschoben. Wir hätten es natürlich genausogut in einen besonderen Speicherbereich bringen können:

Übung 6.10:
Ändern Sie das Programm von oben so ab, daß die in WORT zusammengefaßte Eingabe in einem mit BASIS beginnenden Speicherbereich abgelegt wird.

Übung 6.11:
Modifizieren Sie das Programm so, daß die Übernahme beendet wird, sobald das Zeichen „S" im Datenstrom entdeckt worden ist.

Hardware-Alternative

Wie bei den meisten Standardalgorithmen zur Ein- und Ausgabe ist es auch hier möglich, diese Funktion durch Hardware ausüben zu lassen. Dazu dient ein UART genannter Baustein, der automatisch die Bits übernimmt und sie zu Worten zusammenfaßt. Wenn man jedoch die Bauelementzahl niedrig halten will, so wird man das vorgestellte Programm an seiner Stelle verwenden müssen.

Übung 6.12:
Ändern Sie das Programm so ab, daß die Daten von Bit 0 der Speicherstelle EINGABE und der Status von Bit 0 der Speicherstelle EINGABE x 1 übernommen werden.

Zusammenfassung zu den E/A-Grundlagen

Wir haben damit gelernt, wie man die Grundoperationen zur Datenein- und -ausgabe durchführt und wie man mit einem Strom von parallelen oder seriellen Daten umgeht. Damit sind wir so weit, mit Ein/Ausgabeeinheiten Kontakt aufnehmen zu können.

Datenverkehr mit Ein/Ausgabeeinheiten

Um Daten mit Ein- oder Ausgabeeinheiten austauschen zu können, müssen wir zunächst klarstellen, ob überhaupt Daten zum Übernehmen vorhanden sind, bzw. ob die betrachtete Einheit bereit ist, auszusendende Daten entgegenzunehmen. Dazu können zwei Vorgehensweisen dienen: Quittungsbetrieb (handshaking) und Programmunterbrechungen (interrupts).

Quittungsbetrieb

Quittungsbetrieb setzt man üblicherweise zum Datenverkehr zwischen zwei asynchronarbeitenden Geräten ein, d.h. zwischen zwei Geräten, die nicht gleichzeitig gleichartige Aufgaben wahrnehmen können, nicht synchron arbeiten. Wenn wir z.B. an einen parallel gesteuerten Drucker ein Wort senden möchten, dann müssen wir zuerst sicherstellen, daß der Eingabepuffer des Druckers ein Zeichen übernehmen kann. Das heißt, wir haben den Drucker zu fragen, ob er zur Datenübernahme bereit ist. Dieser antwortet darauf mit „Ja" oder „Nein". Ist er nicht bereit, müssen wir warten. Ist er bereit, so können wir die Daten senden (vgl. Bild 6-8).

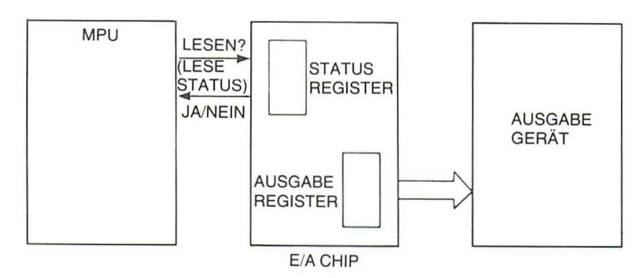


Bild 6-8: Quittungsbetrieb bei der Ausgabe

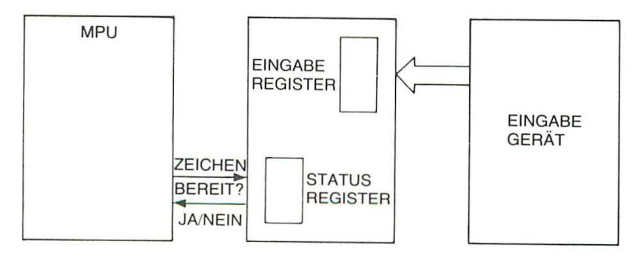


Bild 6-9: Quittungsbetrieb bei der Eingabe

Wollen wir umgekehrt von einem Eingabegerät Daten übernehmen, so müssen wir zunächst fragen, ob die vorgefundenen Bitmuster gültige Daten darstellen. Wir fragen: „Sind die Daten gültig?" Und das Gerät antwortet mit „Ja" oder „Nein". Dieses „Ja" oder „Nein" kann durch Statusbits oder sonst geeignet übermittelt werden (siehe Bild 6-9).
Kurz gesagt: Immer wenn man Information mit einem Gerät austauschen möchte, das unabhängig vom anderen arbeitet und möglicherweise gerade mit etwas anderem beschäftigt ist, so muß vorher festgestellt werden, daß dieses zum Datenverkehr bereit ist. Man läßt sich die Bereitschaft „quittieren". Der eigentliche Datenaustausch folgt darauf. Diese Vorgehensweise wird bei Ein/Ausgabeeinheiten in der Regel angewendet.
Verdeutlichen wir das anhand eines einfachen Beispiels:

Ein Zeichen an einen Drucker senden

Nehmen wir an, in Speicherstelle ZEICHEN stünde ein an den Drucker zu übergebendes Zeichen. Das Programm, es wirklich drucken zu lassen, erhält dann folgende Form:

```
DRUCKEN  LDX  ZEICHEN   ZEICHEN VOM SPEICHER HOLEN
WARTEN   LDY  STATUS    DRUCKER BEREIT? (BIT 7)
         BPL  WARTEN    NEIN: WARTEN
         TXA            SONST ZEICHEN UEBER AKKU
         STA  DRUCKER   AN DRUCKER SENDEN
```

Zuerst laden wir Register X mit dem auszudruckenden Zeichen aus dem Speicher. Dann testen wir das Statusbit des Druckers daraufhin, ob er zur Zeichenübernahme bereit ist. Solange das nicht der Fall ist, bleiben wir in einer Warteschleife. Der Drukker zeigt seine Übernahmebereitschaft durch eine „1" an Bit 7 in STATUS an (das sogenannte „READY"-[Bereitschafts]-Bit). Ist er bereit, so holen wir das in X festgehaltene Zeichen in den Akkumulator.

```
         TXA
```

und geben es an das Ausgaberegister für den Drucker, „DRUCKER", weiter:

```
         STA  DRUCKER
```

Übung 6.13:
Ändern Sie das Programm so ab, daß eine Kette von N Zeichen (N kleiner als 256) gedruckt wird.

Übung 6.14:
Ändern Sie das Programm für den Ausdruck einer Zeichenkette, bis eine „Wagenrücklauf" (carriage return, CR, ASCII 0D) ausgegeben worden ist.

Verkomplizieren wir die Ausgabearbeit jetzt etwas, indem wir eine Kodeumwandlung bei der Ausgabe fordern und indem man mehrere Einheiten zugleich ausgeben muß.

Ausgabe an eine 7-Segment-Anzeige

Eine traditionelle 7-Segment-Anzeige (üblicherweise aus Leuchtdioden, LEDs, aufgebaut) kann die Ziffern „0“ bis „9“ wiedergeben, ja sogar die Hexadezimalziffern „0“ bis „F“, indem bestimmte Kombinationen ihrer in Form einer Acht angebrachten 7 Leuchtbalken (die Segmente) zum Leuchten gebracht werden. Eine solche 7-Segment-LED ist in Bild 6-10 wiedergegeben. Die darzustellenden Zeichen finden sich in Bild 6.11. Man zählt die LED-Segmente von oben angefangen im Uhrzeigersinn von „A“ bis „G“ durch, wobei das G-Segment den Mittelbalken darstellt. (Bild 6-10).

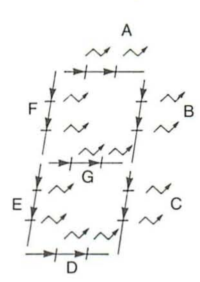


Bild 6-10: Eine Siebensegment-LED-Anzeige

So stellt man z. B. eine „0“ durch Ansteuern der Segmente „ABCEEF“ dar. Nehmen wir nun an, Bit 0 eines Ausgabetors sei mit Segment „A“ verbunden, Bit 1 mit Segment „B“ usw. Bit 7 sei unbenutzt. Damit lautet der binäre Kode zur Darstellung einer „0“: „00111111“, hexadezimal „3F“.

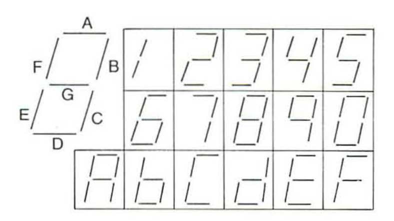


Bild 6-11: Hexadezimalziffern erzeugt mit einer Siebensegmentanzeige

Übung 6.15:
Ermitteln Sie die hexadezimalen Ansteuerungskodes für die Hexadezimalzahlen „0“ bis „E“. Füllen Sie dazu die folgende Tabelle aus (siehe nächste Seite):

Hex	Kode	Hex	Kode	Hex	Kode	Hex	Kode
0	3F	4		8		C	
1		5		9		D	
2		6		A		E	
3		7		B		F	

Stellen wir nun mit *mehreren* 7-Segment-Anzeigen Hexadezimalzahlen dar.

Ansteuerung mehrstelliger 7-Segment-Anzeigen

Eine LED besitzt keinen Speicher. Sie zeigt die Daten nur so lange an, wie sie angesteuert wird. Um die Kosten für eine mehrstellige LED-7-Segment-Anzeige niedrig zu halten, muß der Mikroprozessor die jeweilige Information der einzelnen Stellen *nacheinander für jede Stelle* ausgeben. Dieses wiederholte Ansteuern muß so schnell vor sich gehen, daß das Auge kein Blinken oder Flimmern bemerkt. Das bedeutet, daß pro Anzeigestelle weniger als 100 Millisekunden zur Verfügung stehen. Entwerfen wir ein Programm, das diese Aufgabe übernehmen kann. Register Y soll die Anzeigestelle bezeichnen, die als nächste an der Reihe ist. Zuerst müssen wir die auszugebende Hexadezimalzahl in ihre 7-Segment-Entsprechung umwandeln. Die Umwandlungstabelle dazu haben wir im vorigen Abschnitt erstellt. Da wir eine Tabelle bearbeiten, werden wir indizierte Adressierung benutzen, wobei der Abstand von der Tabellenspitze durch den auszugebenden Hexadezimalwert als Index gegeben ist. Beispielsweise erhalten wir die 7-Segment-Darstellung für die Ziffer „3“ indem wir das vierte Tabellenelement übernehmen. Das ist aber gerade das Element an der Stelle „Tabellenspitze + 3“. Nennen wir die Adresse des Anfangs der 7-Segment-Tabelle „SEGBAS“. Dann kommen wir zu folgendem Programm:

```
LEDANST  TAX                 INDEX: HEXADEZIMAL
         LDA   SEGBAS,X      7-SEGMENT-FORM HOLEN
         STA   SEGDAT        MUSTER AUSGEBEN
         STY   SEGDAR        UND ANZEIGE AKTIVIEREN
         LDX   # $70         VERZOEGERUNGSZEIT SETZEN
VERZG    DEX                 VERZOEGERUNG
         BNE   VERZG         DURCHFUEHREN
         DEY                 NAECHSTE ANZEIGE WAEHLEN
         BNE   FERTIG        NICHT 0: RUECKKEHR
         LDY   STELLEN       SONST ZEIGER NEU SETZEN
FERTIG   RTS                 ZURUECK ZUM HAUPTPROGRAMM
```

Die Anzeige benutzt zwei Register zur Ansteuerung. Das eine, die Segmentdaten SEGDAT, erhält das anzuzeigende Bitmuster, wie es sich aus der Umwandlungstabelle ergibt. Das andere, die Segmentadresse SEGDAR, erhält die Nummer der Stelle, die angesteuert werden soll. Dabei ist angenommen, daß die Anzeige immer dann eingeschaltet wird, wenn man die zugehörige Nummer in SEGDAR schreibt. Das Programm setzt voraus, daß Register Y die Stellennummer bei Aufruf bereits enthält und daß die anzuzeigende Hexadezimalziffer in Register A übergeben wird.

Das Programm holt zunächst aus der Umwandlungstabelle unter SEGBAS das der auszugebenden Ziffer entsprechende Bitmuster in den Akkumulator. Dabei dient Register X als Index für den Tabellenzugriff. Die beiden darauffolgenden Befehle schalten dann die zugehörigen Segmente in der gültigen Anzeigenstelle ein.
Es folgt eine Verzögerung aus drei Befehlen, bevor zur Auswahl der nächsten Anzeigenstelle übergegangen wird, was durch Dekrementieren des Anzeigenregisters Y geschieht.
Hier sind zwei Fälle zu unterscheiden. Wird Y auf einen Wert ungleich Null heruntergezählt, so ist alles in Ordnung. Im andern Fall muß Y wieder auf die höchste gültige Anzeigenstelle gesetzt werden. Danach kann mit RTS (return from subroutine – aus dem Unterprogramm zurückkehren) zum aufrufenden Programm zurückgegangen werden.

Übung 6.16:
Wenn das obenstehende Programm als Unterprogramm eingesetzt wird, so verändert es den Inhalt der beiden Indexregister X und Y. Nehmen Sie an, das Unterprogramm könne den durch die Adressen T1, T2, T3, T4 und T5 gegebenen Speicherbereich frei benutzen. Fügen Sie dann am Programmanfang und am Ende Befehle an, die garantieren, daß die Inhalte von X und Y bei Rückkehr zum Hauptprogramm dieselben wie bei Aufruf des Unterprogramms sind.

Übung 6.17:
Führen Sie dieselbe Aufgabe durch, diesmal jedoch unter der Annahme, daß der durch T1 usw. gegebene Speicherbereich nicht verfügbar ist. (Hinweis: Denken Sie daran, daß der Mikroprozessor einen Mechanismus eingebaut hat, mit dessen Hilfe man Informationen chronologischer Reihenfolge speichern kann.)

Damit haben wir die allgemeinen Ein/Ausgabeprobleme gelöst. Wenden wir uns also einer realen Peripherieeinheit zu: dem 8-Kanal-Fernschreiber (Teletype).

Teletype-Ein/Ausgabe

Die Teletype ist eine serielle Ein/Ausgabeeinheit. Sie sendet und empfängt Worte in serieller Form. Jedes Zeichen ist in ASCII-Form kodiert (siehe die ASCII-Tabelle im Anhang). Zusätzlich wird jedes Zeichen durch ein „Startbit“ eingeleitet und durch zwei „Stopbits“ abgeschlossen. In der gebräuchlichsten Anschlußart, der sogenannten 20-Milliampere-Stromschleife hat die Leitung im Ruhezustand den Wert „1“ (es fließt ein Strom von 20mA). Man benutzt das, um Leitungsunterbrechungen feststellen zu können, bei denen kein Strom mehr fließen würde (= log. „0“). Ein Starbit hat dann den Wert „0“. Es zeigt der angeschlossenen Empfangseinheit an, daß acht Datenbits folgen. Die Standardteletype arbeitet mit einer Geschwindigkeit von 10 Zeichen pro Sekunde. Da wir wissen, daß für jedes Zeichen 11 Bits gebraucht werden, ergibt das eine Übertragsgeschwindigkeit von 110 Bits pro Sekunde. Ein Bit pro Sekunde wird auch „Baud“ genannt, so daß man diesen Fernschreibertyp auch als 110-Baud-Einheit bezeichnet. Lassen Sie uns ein Programm entwickeln, das den seriellen Datenverkehr mit einer Teletype bei korrekter Geschwindigkeit abwickeln kann.

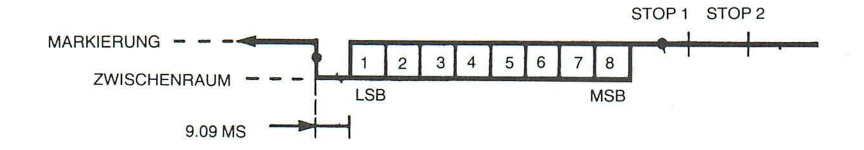


Bild 6-12: Das Teletype-Übertragungsformat

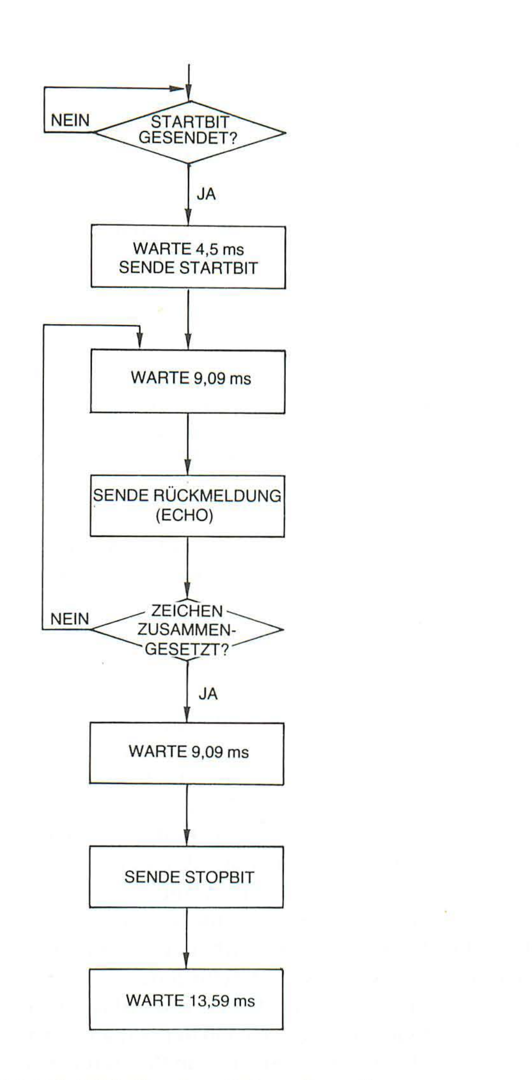


Bild 6-13: TTY-Eingabe mit Echo-Operation

110 Bits pro Sekunde bedeutet, daß für jedes Bit 9,09 Millisekunden zur Verfügung stehen. So lange muß also unsere Verzögerung zwischen aufeinanderfolgenden Bits im Datenverkehr dauern. Das Format eines Teletypeworts steht in Bild 6-12, das Flußdiagramm für Eingabeoperationen finden Sie in Bild 6-13. Das dazugehörige Programm ist in Bild 6-14 wiedergegeben.

```
TTYN   LDA  STATUS
       BPL  TTYIN   STATUSABFRAGE
       JSR  VERZOE  9,09 MS WARTEN
       LDA  TTYBIT  START BIT
       STA  TTYBIT  RUECKMELDUNG
       JSR  VERZOE
       LDX  #$08    BITZAEHLER
NAECH  LDA  TTYBIT  EINGABE SICHERN
       STA  TTYBIT  RUECKMELDUNG
       LSR  A       UEBERTRAG SICHERN
       ROL  ZEICHE  ZEICHEN SICHERN
       JSR  VERZOE
       DEX          NAECHSTES BIT
       BNE  NAECH
       LDA  TTYBIT  STOP BIT
       STA  TTYBIT
       JSR  VERZOE
       RTS
```

Bild 6-14: Programm zur Eingabe durch eine Teletype

Beachten Sie, daß sich das Programm vom Flußdiagramm in Bild 6-13 etwas überschneidet.

Dieses Programm sollte aufmerksam untersucht werden. Seine Logik ist recht einfach. Neu ist nur die Tatsache, daß jedes von der Teletype über TTYBIT übernommene Bit an diese zurückgeschickt und so das zugehörige Zeichen ausgedruckt wird. Das ist ein allgemeines Vorgehen beim Teletypeanschluß. Immer wenn man eine Taste drückt, wird die Information zum Prozessor gesendet und kehrt von dort zum Druckmechanismus des Fernschreibers zurück. Das stellt sicher, daß die Übertragungsleitungen in Ordnung sind und der Prozessor richtig arbeitet, da er das empfangene Zeichen korrekt auf dem Papier abdruckt.

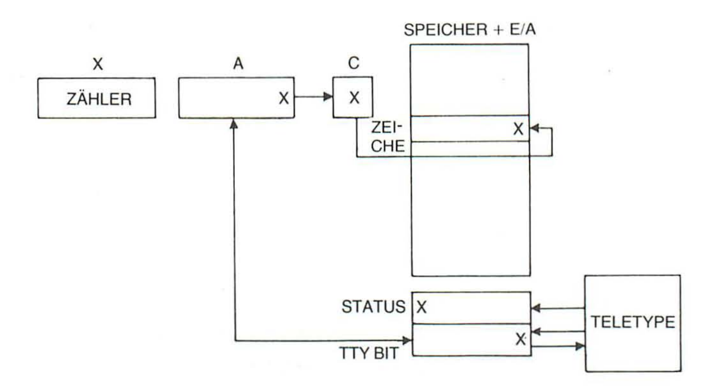


Bild 6-15: Eingabe durch eine Teletype: Registerbelegung

Die ersten beiden Befehle bilden eine Warteschleife. Das Programm wartet, bis das Statusbit signalisiert, daß der Fernschreiber sendefertig ist. Wie gewohnt soll das Statusbit über Bitstelle 7 angeschlossen sein, da diese – das Vorzeichenbit – mit einem einfachen BPL-Befehl (branch on plus) getestet werden kann.
JSR ist ein Unterprogrammaufruf (jump to subroutine – zum Unterprogramm springen). Wir benutzen für die Verzögerung (delay) von 9,09 Millisekunden das Unterprogramm VERZOE. Beachten Sie, daß VERZOE eine Softwareschleife sein kann oder ein Hardwarezähler, falls unser System über einen solchen verfügt.
Als erstes trifft das Startbit ein. Es sollte an die Teletype zurückgesendet, aber ansonsten ignoriert werden. Hierfür sind die Befehle Nummer 4 und 5 da. Beachten Sie, daß die Speicherstelle TTYBIT doppelte Funktion hat, je nachdem, ob wir aus ihr lesen oder in sie hineinschreiben. Beim Lesen übernehmen wir das empfangene Datenbit in Bit 0, beim Schreiben senden wir über dieselbe Bitstelle ein Datenbit an die Maschine aus. Die Hardware hinter TTYBIT gewährleistet, daß diese Funktionen nicht durcheinander kommen.
Dann warten wir auf das nächste Bit. Diesmal handelt es sich um ein aufzubewahrendes Datenbit. Da alle Schiebebefehle das Übertragsbit C mit einbeziehen, brauchen wir zwei Befehle, um unser Datenbit (X in Bild 6) mit LSR A zuerst in C und dann mit ROL in Speicherstelle ZEICHE zu verschieben.
Beachten Sie, daß ROL den Inhalt von C verändert. Wenn wir das Datenbit an den Drucker zurückschicken wollen, müssen wir das tun (STA TTYBIT), bevor es in ZEICHE verschwindet. Darauf warten wir auf das nächste Datenbit (JSR VERZOE) bis wir acht Stück davon zusammen haben (DEX).
Wenn Register X auf Null heruntergezählt worden ist, stehen alle 8 Bits in CHAR. Wir müssen nur noch die Stopbits zurücksenden und sind dann mit der Zeichenübernahme fertig.

Übung 6.18:
Schreiben Sie ein Unterprogramm VERZOE, das eine Verzögerung von 9,09 Millisekunden ergibt.

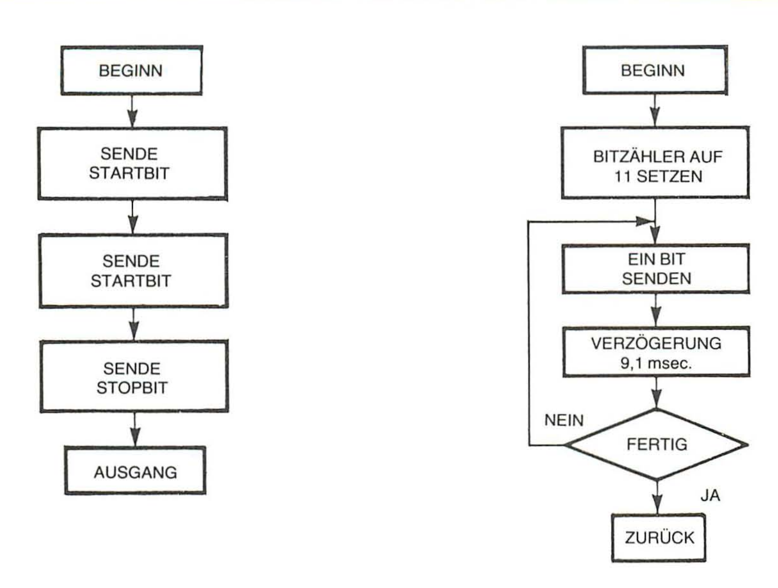


Bild 6-16: Ausgabe an eine Teletype: Flußdiagramme

Übung 6.19:
Schreiben Sie anhand des Beispiels zur Zeicheneingabe ein Programm PRINTC, das an eine Teletype den Inhalt des Akkumulators unter Einsatz von Speicherstelle ZEICHE ausgibt.

Übung 6.20:
Ändern Sie das Eingabeprogramm so ab, daß es auf ein Startbit anstatt auf ein Statusbit wartet.

Ausdrucken einer Zeichenkette

Nehmen wir an, daß das Unterprogramm PRINTC (siehe Übung 6.19) das Ausgeben eines Zeichens auf einen Fernschreiber, ein Bildschirmgerät oder sonst eine Ausgabeeinheit übernimmt. Dieses Programm wollen wir benutzen um den Inhalt des von START bis START + N reichenden Speichers auszudrucken. Dabei werden wir natürlicherweise indizierte Adressierung verwenden. Ansonsten dürften keine Probleme auftreten:

```
DRUCKEN      LDX  #N          WORTANZAHL
NAECHSTES    LDA  START,X     ZEICHEN HOLEN
             JSR  PRINTC      UND AUSDRUCKEN
             DEX              ALLES AUSGEGEBEN?
             BPL  NAECHSTES   NEIN: SCHLEIFE
             RTS              SONST ZURUECK ZUM HAUPT-
                              PROGRAMM
```

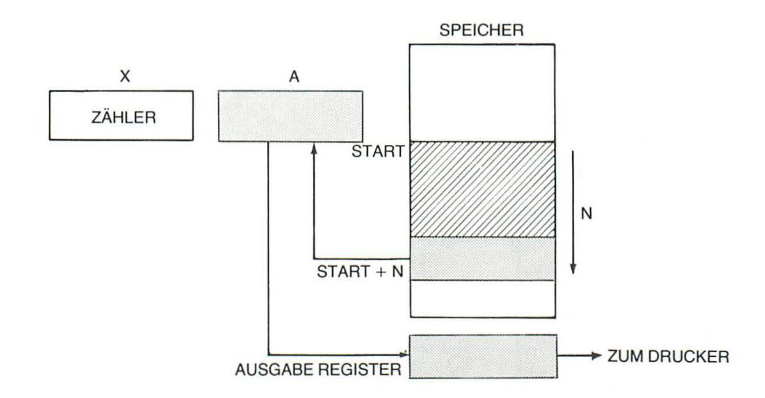


Bild 6-17: Ausdrucken eines Speicherblocks: Registerbelegung

Zusammenfassung zu den Peripherieprogrammen

Damit haben wir die grundlegenden Programmiertechniken beschrieben, die zum Verkehr mit üblichen Ein/Ausgabeeinheiten notwendig sind. Zusätzlich zum Datenverkehr ist es wichtig, ein oder mehrere Steuerregister in den E/A-Bausteinen zu bedienen, um so richtige Übertragungsgeschwindigkeiten, den Unterbrechungsmechanismus und eine Reihe anderer Möglichkeiten vorzugeben. Dazu ist jeweils das Handbuch der betreffenden Einheit zu Rate zu ziehen. (Weitere Einzelheiten zu Algorithmen für den Informationsaustausch mit allen gebräuchlichen Peripherieeinheiten finden Sie in den „Mikroprozessor Interface Techniken".)
Wir haben damit gelernt, wie man mit einzelnen Geräten umgeht. In der Praxis sind an die Busse mehrere Geräte angeschlossen, die unter Umständen gleichzeitig bedient werden wollen. Die Frage stellt sich, wie man dabei die vom Prozessor verfügbare Zeit verwaltet.

Verwaltung von Ein- und Ausgabe

Da mehrere Ein- oder Ausgabeoperationen gleichzeitig notwendig werden können, muß irgendein Verwaltungsmechanismus im System vorhanden sein, der festlegt, welche Bedienungsanforderung zuerst erfüllt wird und wann das zu geschehen hat. Es sind dabei drei Grundtechniken in Gebrauch, die auch kombiniert werden können: Abfragetechnik (polling), Programmunterbrechung (interrupt) und direkter Speicherzugriff (direct memory access, DMA). Wir werden hiervon Abfragen und Unterbrechungen beschreiben. DMA ist eine reine Hardwareangelegenheit, die uns hier nicht betrifft. (Sie finden eine ausführliche Darstellung davon in den „Mikroprozessor Interface Techniken".)

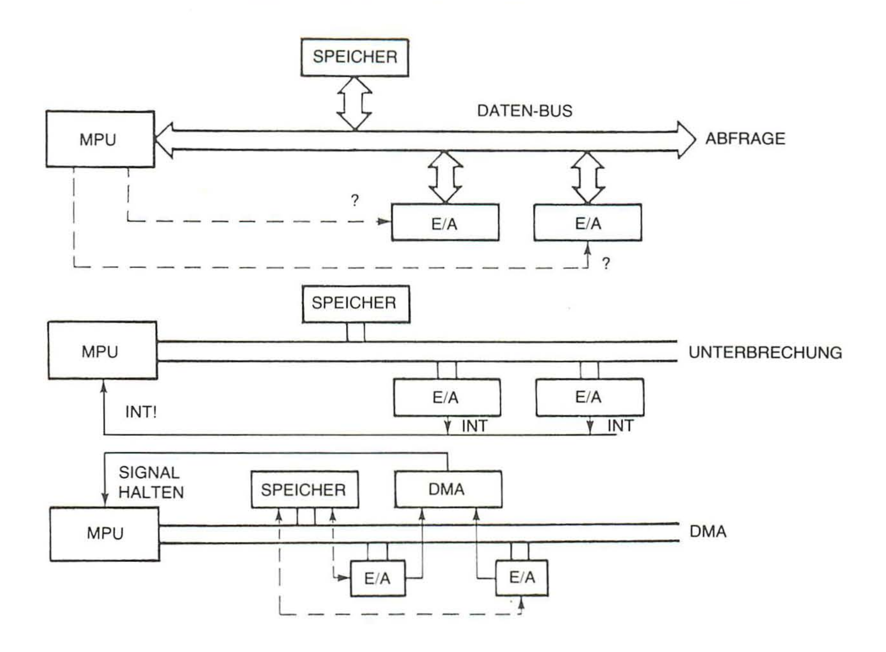


Bild 6-18: Die drei Methoden zur E/A-Steuerung

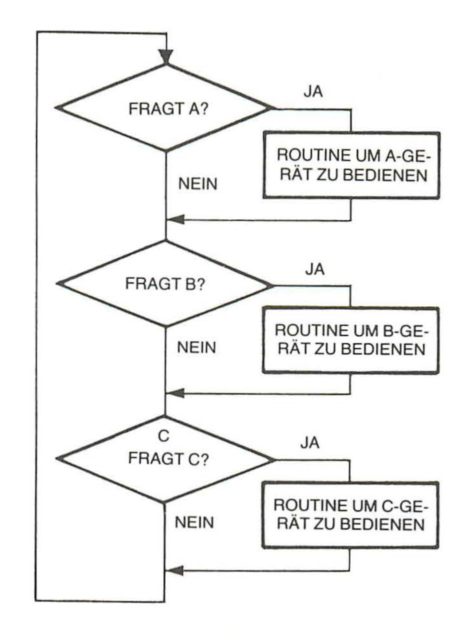


Bild 6-19: Eine Abfrageschleife

Abfragemethode (polling)

Die Abfragemethode ist die einfachste unter den Möglichkeiten, mehrere Peripherieeinheiten zu verwalten. Bei dieser Verfahrensweise fragt der Prozessor die an das System angeschlossenen Einheiten der Reihe nach ab, ob eine Bedienung notwendig ist. Sollte dies der Fall sein, so wird das zugehörige Programm ausgeführt. Wenn nicht, wird die nächste Einheit abgefragt. Derartige Abfragetechniken können nicht nur zur Bedienung externer Einheiten sondern *für jede im System notwendige Bedienungsaufgabe* durchgeführt werden.

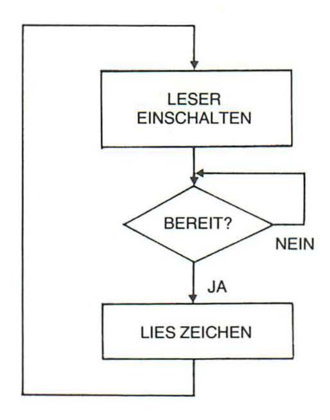


Bild 6-20: Datenübernahme von einem Lochstreifenleser

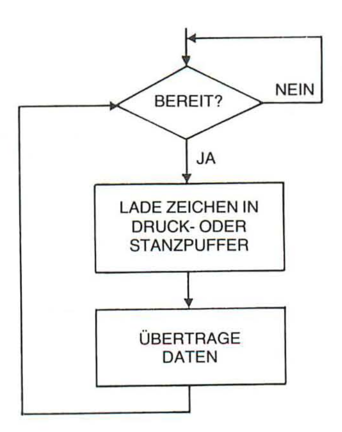


Bild 6-21: Datenausgabe an einen Lochstreifenstanzer

Wenn das System z.B. mit einer Teletype, einem Bandgerät und einer Bildschirmstation ausgestattet ist, so wird das Abfrageprogramm die Teletype fragen: „Willst Du ein Zeichen übergeben?“ Sie wird das Unterprogramm zur Zeichenausgabe an die Teletype fragen: „Muß ein Zeichen gesendet werden?“ Sind die Antworten darauf negativ, so werden die Unterprogramme für das Tonbandgerät abgefragt und schließlich die Bildschirmanzeige. Wenn nur ein Gerät an das System angeschlossen ist, so läßt sich die Abfragetechnik ebenso einsetzen, um herauszufinden, ob eine Bedienung notwendig ist. Ein Beispiel sind die Flußdiagramme in Bild 6-20 und 6-21, die einen Lochstreifenleser bzw. einen Drucker bedienen sollen.
Betrachten wir für ein Beispielprogramm die Abfrageschleife für drei Geräte aus Bild 6-19:

```
ABFRAGE  LDA  STATUSA   GERAET A BEDIENEN? (BIT 7)
         BPL  ABFR1     NEIN: NAECHSTES GERAET
         JSR  GERAETA   SONST GERAET A BEDIENEN
ABFR1    LDA  STATUSB   GERAET B BEDIENEN?
         BPL  ABFR2     NEIN: NAECHSTES GERAET
         JSR  GERAETB   SONST GERAET B BEDIENEN
ABFR2    LDA  STATUSC   GERAET C BEDIENEN?
         BPL  ABFRAGE   NEIN: VON VORNE ANFANGEN
         JSR  GERAETC   SONST GERAET C BEDIENEN
         JMP  ABFRAGE   WIEDER VON VORNE ANFANGEN
```

Bit 7 der Statusregister der verschiedenen Geräte ist eine „1“, wenn sie bedient zu werden wünschen. Ist dies der Fall, springt die Schleife in ein Unterprogramm, in dem die jeweilige Bedienungsarbeit ausgeführt wird und kehrt dann zur Abfrage des nächsten Geräts zurück. Muß ein Gerät nicht bedient werden, so überspringt das Abfrageprogramm einfach den Aufruf der Bedienung und fragt die nächste Einheit ab.
Die Vorteile der Abfragetechnik sind offensichtlich: Sie ist einfach, braucht keine Hardwareunterstützung (außer den Statusregistern) und hält die Gerätebedienung mit dem übrigen Programm synchron. Doch genauso offenbar sind die Nachteile der Methode: Der Prozessor verschwendet sehr viel Zeit damit, Einheiten abzufragen, die gar keine Bedienung benötigen. Mehr noch, wenn eine Einheit rasch bedient werden muß, kann es sein, daß der Prozessor bei der Abfragetechnik zu spät kommt und so wertvolle Information verlorengeht.
Daher ist ein anderer Mechanismus wünschenswert, der garantiert, daß die Arbeitszeit des Prozessors für sinnvolle Operationen anstatt für unnötige Bedienungsabfragen verwendet werden kann. Trotzdem muß betont werden, daß die Abfragetechnik immer dann vorzuziehen ist, wenn der Prozessor ohnehin nichts anderes zu tun hat, denn diese Technik hält die Gesamtorganisation einfach. Doch sehen wir uns jetzt die Hauptalternative zur Abfragetechnik an: die Programmunterbrechungen.

Programmunterbrechungen (interrupts)

Das Prinzip von Programmunterbrechungen zeigt Bild 6-18. Es gibt eine besondere Steuerleitung, die „interrupt“-Verbindung, die an einen speziellen Mikroprozessoranschluß gelegt ist. An diese Leitung können mehrere Ein/Ausgabegeräte ange-

schlossen sein. Benötigt irgendeine von ihnen eine besondere Bedienung durch das System, so sendet sie einen Impuls über die Leitung oder legt sie auf einen bestimmten Pegel: Sie fordert eine Programmunterbrechung an (interrupt request). Sehen wir uns an, wie der Prozessor auf eine solche Unterbrechungsanfrage reagiert (Bild 6-22).

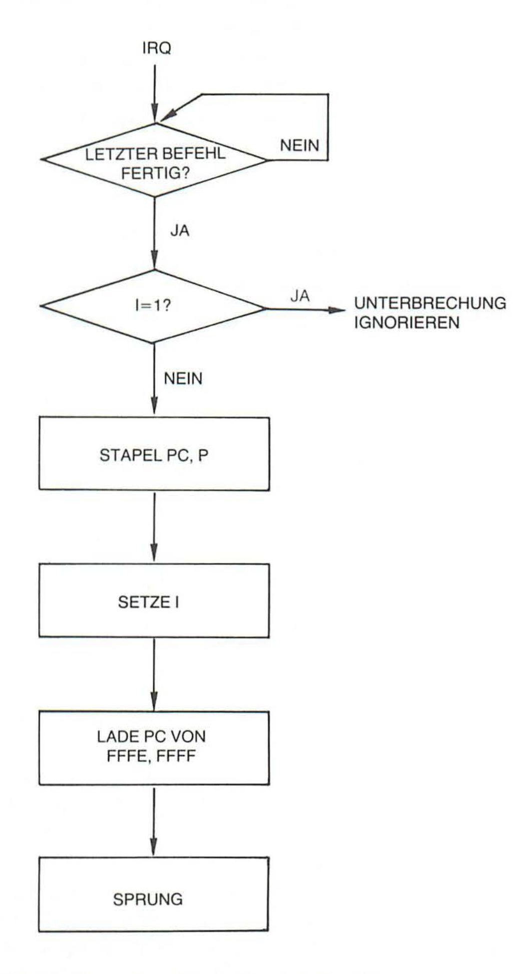


Bild 6-22: Abarbeitung einer Programmunterbrechung

Zunächst beendet er auf alle Fälle den gerade bearbeiteten Befehl, da sonst im System das reine Chaos ausbrechen könnte. Darauf unterbricht er die gerade ausgeführte Arbeit und springt zu einem besonderen Programm, das die angeforderte Bedienung ausführt. Das hat Ähnlichkeit mit dem Aufruf eines Unterprogramms: Da der Prozessor nach Abarbeiten der Unterbrechung zum unterbrochenen Programm zurückkehren muß, hat er vorher den aktuellen Programmzählerstand auf den Stapel zu retten. *Jede Programmunterbrechung speichert den Programmzählerstand auf dem Sta-*

pel! Dazu sollten auch die Flaggen im Statusregister P auf den Stapel gebracht werden, da ihr Inhalt mit Sicherheit während der Programmunterbrechung verändert wird. Wenn schließlich das Unterbrechungsprogramm, das die angeforderte Bedienung ausführt, sonst irgendwelche allgemeinen Register verändert, so muß auch ihr Inhalt vorher auf den Stapel gerettet werden.
Erst nachdem alle diese Registerinhalte sicher untergebracht sind, kann man zum angeforderten Bedienungsprogramm verzweigen. Nach dessen Bedienung müssen die Register wieder auf ihren Stand vor der Unterbrechung gebracht und dann mit einem Rücksprungbefehl die Arbeit an der unterbrochenen Stelle wieder aufgenommen werden. In allen Fällen ist ein „Rücksprung aus einer Unterbrechung" (return from interrupt) vorgesehen, der neben dem Programmzähler auch automatisch das Statusregister wieder in seinen alten Stand setzt. Betrachten wir dies jetzt für den 6502-Prozessor im Einzelnen.

Programmunterbrechung beim 6502

Der 6502-Prozessor besitzt zwei Eingänge zur Anforderung einer Programmunterbrechung: IRQ (interrupt request) und NMI (non-maskable interrupt). IRQ ist der normale Unterbrechungsanschluß, während NMI eine höhere Priorität besitzt und nicht ausmaskierbar ist. Sie arbeiten wie folgt:

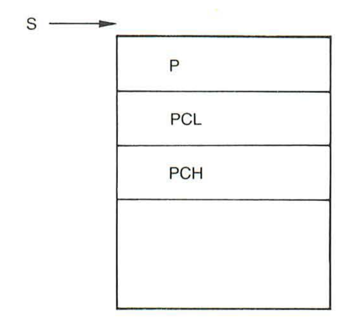


Bild 6-23: Der 6502-Stapel nach Einleiten einer Programmunterbrechung

IRQ wird durch einen bestimmten Spannungspegel aktiviert. Sein Zustand wird durch den Mikroprozessor während der Programmausführung überwacht, wenn die Unterbrechungsmaske I im Statusregister nicht gesetzt ist. Andernfalls bleiben etwaige Unterbrechungsanforderungen am IRQ-Anschluß unbeachtet. Nehmen wir an, daß Programmunterbrechungen möglich sind (interrupt enabled). Ist IRQ aktiv (I = Q), so testet der Prozessor *nach jeder Befehlsausführung* den Statuspegel am IRQ-Anschluß. Wird dort eine Unterbrechungsanforderung vorgefunden, so wird automatisch die Unterbrechungsmaske im Statusregister auf „1" gesetzt. Das verhindert, daß der Prozessor ein zweitesmal unterbrochen werden kann und gibt ihm so Gelegenheit, zunächst einmal in Ruhe die wichtigsten Arbeitsregister auf den Stapel zu retten. Nach Setzen der I-Maske speichert der 6502 automatisch den Inhalt des Programmzählers PC und den des Statusregisters P auf dem Stapel. Die Stapelbelegung nach Entgegennahme einer Unterbrechung zeigt Bild 6-23.

Als nächstes lädt der 6502 automatisch den Inhalt der Speicherzellen FFFE und FFFF in den Programmzähler. Man nennt diese 16-Bit-Speicherstelle demzufolge den „Unterbrechungszeiger" (interrupt vector). D.h. der 6502 verzweigt bei Entgegennahme von IRQ automatisch zu der in FFFE, FFFF angegebenen Stelle. Der Benutzer ist dafür verantwortlich, daß diese Speicherstellen einen sinnvollen Inhalt besitzen. Es können nun aber verschiedene Einheiten an IRQ angeschlossen sein. Der Prozessor springt aber immer nur zu einem bestimmten Unterbrechungsprogramm. Wie soll man die verschiedenen Anforderungen voneinander unterscheiden? Wir werden das im nächsten Unterkapitel betrachten.
Der NMI-Unterbrechungseingang hat prinzipiell dieselbe Aufgabe wie IRQ, mit dem Unterschied, daß er nicht mit der Unterbrechungsmaske I unterdrückt werden kann. Er hat eine höhere Priorität als IRQ, d.h. liegen an beiden Eingängen gleichzeitig Anfragen an, so wird zuerst NMI bedient. Üblicherweise benutzt man ihn für sehr dringende Rettungsoperationen, wie sie z.B. beim Zusammenbruch der Versorgungsspannung eintreten können. Die Arbeitsweise entspricht ansonsten genau der einer normalen Unterbrechung, nur daß hier der Unterbrechungszeiger aus FFFA und FFFB geholt wird. Bild 6-24 verdeutlicht das.

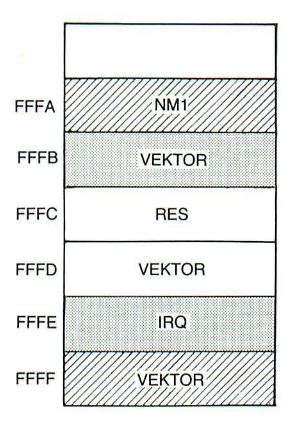


Bild 6-24: Die Unterbrechungszeiger im 6502-System

Die Rückkehr aus der Unterbrechung erfolgt mit dem Befehl RTI (return from interrupt), der die obersten drei Bytes des Stapels in Statusregister P bzw. Programmzähler PC zurück überträgt. Damit kann das unterbrochene Programm seine Arbeit wieder aufnehmen. Der interne Zustand der Maschine ist genau derselbe wie vor der Unterbrechung. Für das unterbrochene Programm ist lediglich eine Verzögerung eingetreten.
Vor der Rückkehr aus einem Unterprogramm muß das Programm allerdings noch ausdrücklich die Unterbrechungsmaske I wieder löschen, sollen weitere Programmunterbrechungen zugelassen werden. Außerdem sind die etwa geretteten Arbeitsregister, insbesondere die Prozessorregister A, X und Y wieder in ihren vorigen Stand zu versetzen, bevor ein RTI-Befehl ausgeführt wird. Andernfalls erhält das unterbrochene Programm die falschen Informationen zurück und kann nicht richtig weiterarbeiten.

Nehmen wir an, das Unterbrechungsprogramm benutze die Register A, X und Y, so sind fünf Befehle nötig, um ihren Inhalt vor allen anderen Arbeiten auf den Stapel zu retten:

```
AXYRETTEN  PHA  A AUF DEN STAPEL
           TXA  REGISTER X HOLEN
           PHA  UND AUF DEN STAPEL DAMIT
           TYA  REGISTER HOLEN
           PHA  UND RETTEN
```

Leider kann der 6502 unmittelbar nur den Akkumulator oder das Statusregister auf den Stapel bringen. Dementsprechend verschlingt das Retten von X und Y Zeit, da hierzu 4 Befehle nötig sind. Bild 6-25 zeigt den Zustand des Stapels nach dieser Operation.

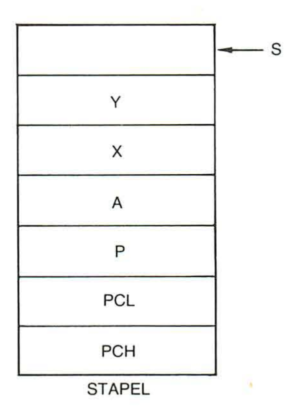


Bild 6-25: Speichern aller Register

Nach Abarbeiten des Unterbrechungsprogramms müssen diese Register wieder zurückgeholt und dann die Unterbrechung beendet werden. Das kann mit den folgenden sechs Befehlen geschehen:

```
PLA  Y VOM STAPEL HOLEN
TAY  UND NEU SETZEN
PLA  X VOM STAPEL HOLEN
TAX  UND SETZEN
PLA  A ZURUECKHOLEN
RTI  UND ZURUECKSPRINGEN
```

Übung:

Berechnen Sie unter Zuhilfenahme der Befehlsausführungszeiten im Anhang die Zeit, die durch das Retten und dann wieder Zurückholen der Register A, X und Y verlorengeht.

Eine grafische Veranschaulichung des Unterschieds zwischen Abfragetechnik (polling) und Programmunterbrechung (interrupt) zeigt Bild 6-18. Dort wird die Abfragemethode an der Spitze dargestellt, der Unterbrechungsprozeß ist darunter verdeutlicht. Man sieht, daß das Programm beim Abfragen viel Zeit durch Wartereien verliert. Beim Einsatz von Unterbrechungen wird das laufende Programm angehalten und die angeforderte Bedienung eingeschoben, wonach die Arbeit wieder aufgenommen werden kann. Ein offensichtlicher Nachteil dieser Methode ist jedoch die Tatsache, daß am Anfang und am Ende der Unterbrechung zusätzliche Befehle notwendig werden können, die eine weitere Verzögerung bewirken, bevor die Bedienung ausgeführt werden kann. Das ist ein systembedingter Mehraufwand.
Nachdem wir damit über die prinzipielle Arbeit der beiden Unterbrechungsanschlüsse Klarheit gewonnen haben, müssen wir noch zwei wichtige Probleme untersuchen:

1. Was macht man, wenn mehrere Einheiten eine Unterbrechung anfordern können?

2. Was ist zu tun, wenn eine Einheit eine Unterbrechung anfordert, während gerade eine andere Programmunterbrechung abgearbeitet wird?

Mehrere Einheiten auf der Unterbrechungsleitung

Wenn eine Programmunterbrechung angefordert wird, dann verzweigt der Prozessor automatisch zu der in FFFE, FFFF (für IRQ) bzw. in FFFA, FFFB (für NMI) angegebenen Adresse. Bevor nun irgendeine Bedienungsarbeit durchgeführt werden kann, muß der Prozessor wissen, welche von den an die Leitung angeschlossenen Einheiten die Unterbrechung bewirkt hat. Dazu gibt es wie üblich zwei Methoden, eine in Hard- und eine in Software.

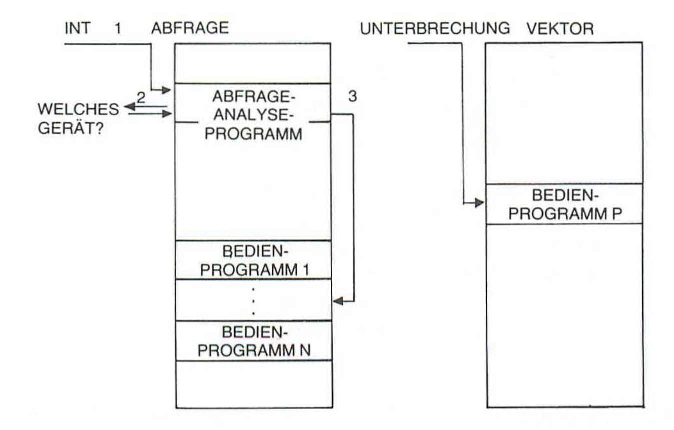


Bild 6-26: Methoden zur Feststellung der unterbrechenden Einheit

Bei der Softwaremethode wird eine Abfragetechnik eingesetzt: Der Prozessor fragt alle in Betracht kommenden Einheiten der Reihe nach ab, ob sie die Unterbrechung ausgelöst haben. Das ist in Bild 6-26 wiedergegeben. Ein Programm für diesen Zweck kann z. B. so aussehen:

```
LDA  STATUS 1    UNTERBRECHUNG VON EINHEIT 1?
BMI  EINHEIT 1   JA: EINHEIT 1 BEDIENEN
LDA  STATUS 2    SONST EINHEIT 2?
BMI  EINHEIT 2   JA: EINHEIT 2 BEDIENEN
..........
```

Die Hardwaremethode braucht zusätzliche Bausteine, stellt dafür aber die Adresse des Unterbrechungsprogramms gleichzeitig mit der Unterbrechungsanforderung zur Verfügung. Man benutzt für diesen Zweck zumeist einen besonderen Baustein, der PIC (priority-interrupt controller – prioritätsorientierte Unterbrechungssteuerung) genannt wird. Ein solcher PIC legt automatisch die Adresse des Unterbrechungsprogramms auf den Datenbus, wenn der 6502 den Unterbrechungszeiger aus FFFE, FFFF holen möchte. Bild 6-26 verdeutlicht diesen Prozeß.
In den meisten Fällen ist die Reaktionsgeschwindigkeit auf eine Unterbrechungsanforderung nicht so wichtig, so daß die billigere Softwaremethode eingesetzt werden kann. Muß eine Einheit allerdings sofort bedient werden, so ist die Hardwarelösung einzusetzen.

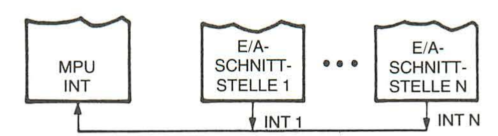


Bild 6-27: Mehrere Einheiten können dieselbe Leistung zur Unterbrechungsanforderung benutzen

Geschachtelte Unterbrechungsanforderungen

Das nächste Problem ist die Tatsache, daß eine neue Unterbrechungsanforderung angefordert werden kann, während eine andere Unterbrechung gerade abgearbeitet wird. Sehen wir uns an, was dabei passiert und wie man den Stapel zur Lösung des Problems einsetzen kann. Betrachten wir dazu Bild 6-28, das das Verhalten bei mehreren Unterbrechungen zeigt. Die zeitliche Aufeinanderfolge der Ereignisse ist von links nach rechts dargestellt. Der untere Teil der Darstellung zeigt den Inhalt des Stapels. Zur Zeit T0, links in der Darstellung wird gerade Programm P ausgeführt. Ein Schritt weiter nach rechts, zum Zeitpunkt T1 wird Unterbrechung I1 angefordert. Wir nehmen an, daß die Unterbrechung freigegeben ist (Maske I = 0). Programm P wird angehalten, was die Unterbrechung des zugehörigen horizontalen Strichs in der Darstellung anzeigt. Der Stapel erhält den Inhalt des Programmzählers und des Statusregisters von Programm P, angedeutet durch den Kasten am Fuß der Abbildung. Zusätzlich können hier alle Register stehen, die vom Unterbrechungsprogramm noch gerettet worden sind.

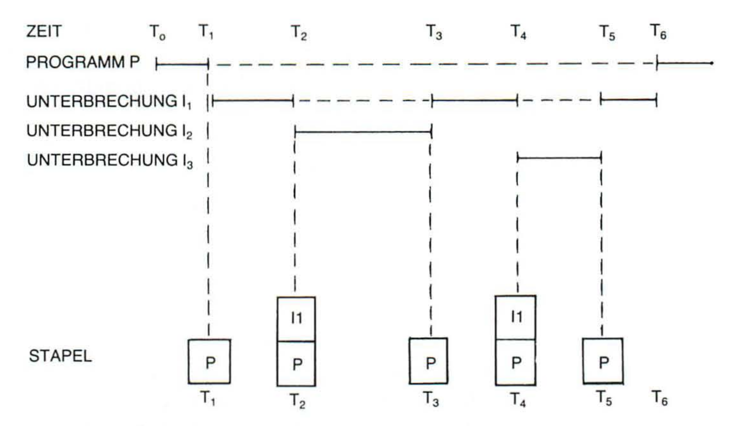


Bild 6-28: Das Stapelverhalten bei Unterbrechungen

Zum Zeitpunkt T1 beginnt Unterbrechungsprogramm I1 seine Arbeit und läuft bis T2, wo eine neue Unterbrechungsanforderung I2 eintrifft. Wir nehmen an, daß auch ihr der Weg freigegeben wird. (Das ist insbesondere der Fall, wenn sie eine höhere Dringlichkeit [Priorität] als I1 besitzt. Andernfalls wird ihre Bearbeitung in den meisten Systemen hinausgeschoben, bis I1 fertig abgearbeitet ist.) Zum Zeitpunkt T2 werden daher die von I1 belegten Register (Programmzähler, Status und evtl. auch andere) auf den Stapel gebracht, was der mit I1 versehene Kasten angibt. I2 nimmt dann seine Arbeit auf, die es zur Zeit T3 abschließt.

Wenn I2 fertig ist, werden die auf den Stapel geretteten Registerinhalte wieder zurückgeholt, wie das die Stapelbelegung zum Zeitpunkt T3 in Bild 6-28 andeutet. Damit nimmt automatisch I1 seine Arbeit wieder auf. Zum Zeitpunkt T4 tritt wieder eine Unterbrechungsanforderung höherer Priorität auf. Wie der untere Bildteil zeigt, werden die Register von I1 wieder auf den Stapel gebracht. Unterbrechung I3 wird von T4 bis T5 abgearbeitet. Bei T5 ist ihre Arbeit beendet, die Registerinhalte von I1 werden in den 6502 zurückgebracht, und I1 setzt seine Arbeit fort, bis sie zum Zeitpunkt T6 damit fertig ist. Dort werden die restlichen auf dem Stapel untergebrachten Register zurückgeholt, so daß das Hauptprogramm P seine Arbeit fortsetzen kann. Wie Sie sehen, ist der Stapel zu diesem Zeitpunkt wieder leer. Die gestrichelten waagerechten Linien in der Abbildung geben zu jedem Zeitpunkt an, wieviele Programmebenen gerade auf dem Stapel liegen.

Übung 6.22:

Nehmen wir an, für jede Unterbrechung würden die drei Register PC, P und A auf den Stapel gebracht. Das belegt dort für jede Unterbrechungsebene vier Speicherplätze. Wie viele Unterbrechungsebenen sind dann möglich. (Denken Sie daran, daß der 6502-Stapel nur 256 Speicherplätze bietet.)

Übung 6.23:

Wieviele Unterbrechungsebenen sind es, wenn zusätzlich Register X und Y gerettet werden müssen? Gibt es andere Faktoren, die die Zahl der möglichen Unterbrechungen noch weiter einschränken?

Es muß allerdings betont werden, daß in der Praxis an ein Mikrocomputersystem normalerweise nur eine relativ geringe Anzahl von Geräten über Unterbrechungsmechanismen angeschlossen sind. Es ist daher recht unwahrscheinlich, daß eine so hohe Zahl von Unterbrechungen in einem solchen System wirklich auftritt.
Wir haben damit alle normalerweise mit Programmunterbrechungen zusammenhängenden Probleme gelöst. Sie sind im Grunde recht einfach zu benutzen, selbst von einem Programmieranfänger. Schließen wir unsere Betrachtung hier mit einer Form von synchroner Programmunterbrechung beim 6502 ab:

Der BRK-Befehl

Der BRK-Befehl des 6502 bewirkt eine Programmunterbrechung durch Software. Man kann ihn in ein Programm einfügen und erhält dann denselben Effekt wie bei einer IRQ-Unterbrechung: PC und P werden auf den Stapel gerettet, und der Prozessor springt zu der in FFFE, FFFF angegebenen Adresse. Dieser Befehl kann sinnvoll zur Fehlersuche in Programmen eingesetzt werden. Er bewirkt, daß das jeweilige Programm an der gewünschten Stelle anhält und zu einem Hilfsprogramm verzweigt, mit dem man den Zustand des Systems analysieren kann. Da IRQ und BRK denselben Effekt haben, ist eine besondere Flagge im Statusregister vorgesehen, die gesetzt wird, sobald eine Softwareunterbrechung durch BRK eingetreten ist (die B-Flagge). Bei einer normalen Unterbrechung ist B = 0. Der Wert dieses Bits kann mit dem folgenden einfachen Programm untersucht werden:

```
BTEST   PLA              P-INHALT VOM STAPEL IN A
        PHA              UND STAPEL ERNEUERN
        AND  # $10       B-BIT ISOLIEREN
        BNE  BRKPRG      WAR BRK: DIES ABARBEITEN
```

Man schließt mit diesem Programm normalerweise das Ende einer Abfragesequenz zur Feststellung einer Unterbrechungseinheit ab.

Warnung:

Bei einem BRK-Befehl wird der Programmzähler *plus 2* gespeichert. Da BRK nur ein Einbytebefehl ist, kann das manchmal zu Schwierigkeiten führen, wenn man den geretteten PC-Wert nicht passend justiert. Der Grund liegt darin, daß man oft einen BRK-Befehl bei der Fehlersuche an Stelle eines anderen Befehls setzt. Die meisten 6502-Befehle umfassen aber zwei Bytes, so daß man sich hier in der Mehrzahl der Fälle unnötige Korrekturarbeiten erspart, indem das Programm nach Rückkehr aus einer BRK-Unterbrechung wieder genau an einer Befehlsgrenze steht.

Zusammenfassung

Wir haben in diesem Kapitel die Vielzahl von Techniken zum Verkehr des Computers mit der Außenwelt vorgestellt. Von einfachen Ein/Ausgabeoperationen bis hin zu komplexeren Problemen haben wir gelernt, wie man hierfür brauchbare Programme erstellt und haben die Leistungsfähigkeit sogenannter „Benchmark"-Programme

im Fall paralleler und serieller Datenübertragung untersucht. Schließlich haben wir gelernt, mehrere Ein/Ausgabeeinheiten mit Hilfe von Abfrage- und Unterbrechungstechniken zu verwalten. Natürlich kann man noch die verschiedensten anderen exotischen Einheiten an ein System anschließen. Mit der Menge der soweit erstellten Techniken und mit einem grundlegenden Verständnis der in Frage kommenden Peripherieeinheiten dürften auch die dabei auftretenden Schwierigkeiten zu lösen sein. Im nächsten Kapitel werden wir die Eigenschaften einiger Ein/Ausgabebausteine untersuchen, die normalerweise zur Verbindung peripherer Geräte mit einem 6502-System dienen.

Übungen:

6.24:
Man kann mit einer 7-Segment-Anzeige auch noch andere Zeichen außer den Hexadezimalziffern wiedergeben. Ermitteln Sie die zur Ansteuerung benötigten Bitmuster für die Buchstaben H, I, J, L, O, P, S, U, Y, g, h, i, j, l, n, o, p, r, t, u, y.

6.25:
Das Flußdiagramm zur Unterbrechungsabarbeitung ist in Bild 6-29 wiedergegeben. Beantworten Sie dazu die folgenden Fragen:
1 – Was geschieht hier durch Hard- und was durch Software?
2 – Zu was benötigt man die Unterbrechungsmaske?
3 – Wieviele Register sind zu retten?
4 – Was macht ein RTI-Befehl? Worin unterscheidet er sich von einer Rückkehr aus einem Unterprogramm (RTS)?
5 – Machen Sie einen Vorschlag, was bei Überlauf des Stapels getan werden könnte.
6 – Wo geht bei Abarbeiten einer Unterbrechung zusätzlich Zeit verloren?

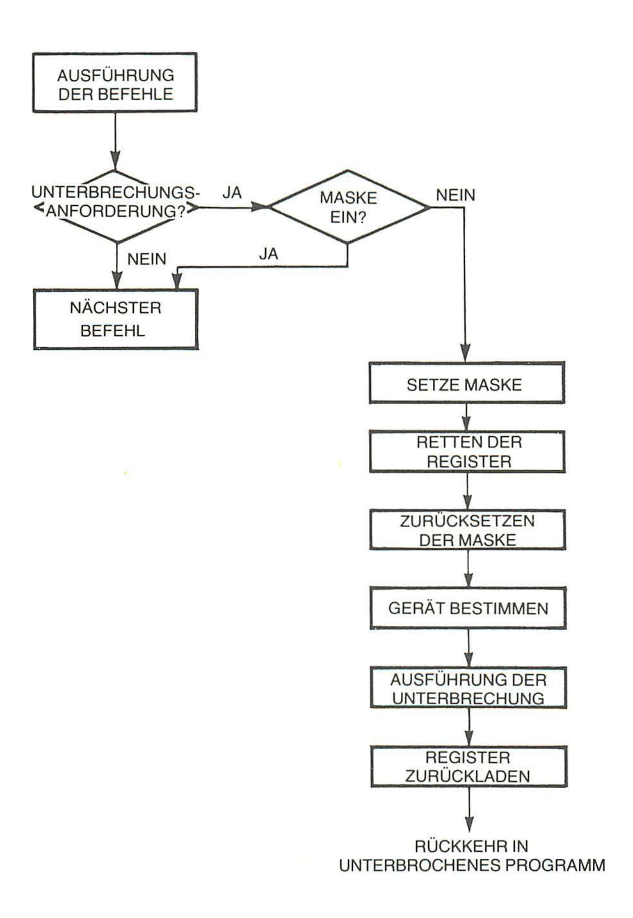


Bild 6-29: Logik der Unterbrechungsbehandlung

KAPITEL 7
EIN/AUSGABEBAUSTEINE

Einführung

Wir haben gelernt, wie man den 6502-Prozessor für die meisten Situationen programmiert. Jedoch sollten wir die Ein/Ausgabebausteine, die man normalerweise in einem 6502-System findet, nicht unerwähnt lassen. Infolge des raschen Fortschritts der LSI-Technik sind neue Bausteinarten auf den Markt gekommen, die es vorher nicht gab. Das führte dazu, daß in einem System außer dem Mikroprozessor selbst *auch noch die verschiedenen Ein/Ausgabebausteine programmiert werden müssen.* Es ist genau betrachtet schwieriger, sich zu merken, wie man diese Bausteine programmiert, als das Programmieren des Prozessors selbst! Das liegt nicht daran, daß die eigentliche Programmierung schwieriger wäre, sondern daß jeder dieser Spezialbausteine seine ganz besonderen Eigenarten hat. Wir wollen hier zuerst den allgemeinsten E/A-Baustein betrachten, einen PIO-Chip (programmable input/output chip – programmierbarer Ein/Ausgabechip), und uns danach die „Verbesserungen“ dieses Standardbausteins vornehmen, die man in 6502-Systemen häufig findet: die Bausteine 6520, 6530, 6522 und 6532. Die vollständigen Eigenschaften dieser Bausteine finden Sie im dem Band „6502 Anwendungen“ vom selben Autor (Nummer D302D).

Der Standard-PIO-Baustein (6520)

Eigentlich gibt es keinen „Standard-PIO-Baustein“. Jedoch entspricht der 6520 in den wesentlichen Funktionen allen ähnlich konstruierten PIOs von anderen Herstellern. Aufgabe eines PIO ist es, eine mehrere Tore umfassende Verbindung zu externen Ein- oder Ausgabeeinheiten herzustellen. (Ein „Tor“ [port] ist nichts weiter als ein Satz von 8 Ein/Ausgabeleitungen). Jeder PIO enthält mindestens zwei Sätze von 8-Bit-Ein/Ausgabeleitungen. Jede E/A-Einheit braucht einen *Datenpuffer,* um den Inhalt des Datenbus zur externen Einheit wenigstens für die Dauer der Ausgabe stabil zu halten. Unsere PIO besitzt so zumindest je einen 8-Bit-Puffer pro Datentor.
Außerdem haben wir im vorigen Kapitel festgelegt, daß der Mikroprozessor in der Regel die Daten im *Quittungsbetrieb* sendet oder empfängt, oder *Programmunterbrechungen* zur Einleitung des Datenverkehrs mit der E/A-Einheit benutzt. Ein PIO verwendet ein ähnliches Verfahren zur Verbindung mit der Peripherieeinheit. Daher muß jeder PIO mindestens *zwei Steuerleitungen pro Datentor* für den Quittungsbetrieb besitzen.

Der Mikroprozessor muß des weiteren den Status jedes Tors lesen können. Jedes Datentor muß daher mit einem oder mehreren *Statusbits* ausgerüstet sein. Schließlich hat jeder PIO eine Reihe verschiedener Betriebsmöglichkeiten. Der Programmierer muß also auf ein besonderes Register im PIO zugreifen können, das ihm gestattet, unter diesen Betriebsmöglichkeiten auszuwählen. Man bezeichnet dieses Register als *Steuerregister.* Beim 6520 ist die Statusinformation Teil des Steuerregisters.
Eine wichtige Eigenschaft des betrachteten PIO ist, daß jede Leitung nach Bedarf zur Ein- oder Ausgabe bestimmt werden kann. Bild 7-1 zeigt ein solches PIO. Der Programmierer kann festlegen, ob eine bestimmte Leitung ein Ein- oder ein Ausgang für Daten sein soll. Zur Festlegung dieser Übertragungsrichtung dient ein sogenanntes *Datenrichtungsregister* (data direction register) bei jedem Tor. Eine „0" in diesem Register legt die Eingabefunktion der zugeordneten Leitung, eine „1" die Ausgabefunktion fest.

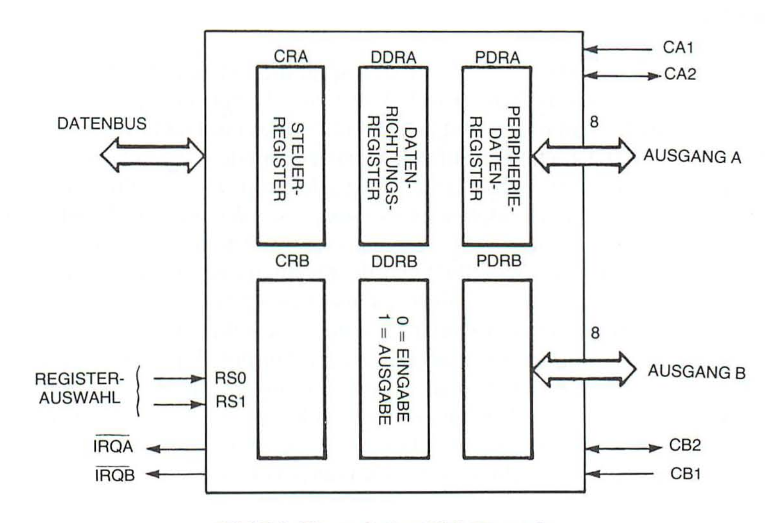


Bild 7-1: Ein typischer PIO-Baustein

Die Wahl dieser logischen Werte, „0" zur Eingabe und „1" zur Ausgabe hat ihren Grund: Wenn das System eingeschaltet wird, ist es sehr wichtig, daß alle Datentore auf *Eingabe* geschaltet sind. Wäre das nicht der Fall, so könnte eine angeschlossene Ausgabeeinheit versehentlich beim Einschalten unkontrolliert loslaufen. Das kann dazu führen, daß sie sich selbst zerstört oder sonst sich und der Umgebung Schaden zufügt. Beim Einschalten des Systems läuft eine automatische Rücksetzfunktion (reset) ab, die alle Einheiten in einen definierten Ausgangszustand bringt. Im Fall eines PIO werden alle internen Register auf „0" gesetzt, die Tore mithin alle zu Eingängen bestimmt.
Der Anschluß des Bausteins an den Mikroprozessor erscheint links in Bild 7-1. Selbstverständlich ist er mit dem 8-Bit-Datenbus verbunden, dazu noch mit dem Adreßbus und den in Frage kommenden Steuerleitungen. Der Programmierer legt einfach eine bestimmte Adresse eines Registers im PIO fest, auf das er zugreifen möchte. Der

6520 (der mit dem 6820 von Motorola identisch ist) besitzt 6 interne Register. Man kann jedoch nur *vier* Register adressieren! Das Problem, auf alle Register zuzugreifen, wird durch Umschalten von Bit 2 im Steuerregister gelöst. Hat dieses den Wert „0“, so wird das jeweilige Datenrichtungsregister adressiert, hat es den Wert „1“, so kann man auf das Datenregister zugreifen. Will der Programmierer demnach das Datenrichtungsregister setzen, so muß er Bit 2 des zugehörigen Steuerregisters löschen, bevor er auf das gewünschte Register zugreifen kann. Das ist etwas umständlich zu programmieren. Man sollte es sich merken, um unnötige Schwierigkeiten auszuschließen.

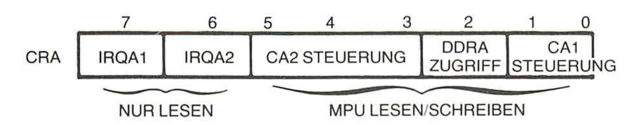


Bild 7-2: Das Steuerwort eines PIA-Bausteins

Die Adressierung der 6520-Register ergibt sich aus der Darstellung in Bild 7-3. Vom Adreßbus werden zwei Auswahlsignale abgeleitet: RS0 und RS1. Durch sie kann der Programmierer eine 2-Bit-Adresse festlegen. CRA ist das Steuerregister für Tor A, CRB für Tor B. CRA (2), bzw. CRB (2) bezeichnet das Umschaltbit 2 in diesen Registern.

RS1	RS0	CRA 2	CRB 2	ausgewähltes Register
0	0	1	–	Peripherie-Register A
0	0	0	–	Datenrichtungs-Register A
0	1	–	–	Steuer-Register A
1	0	–	1	Peripherie-Register B
1	0	–	0	Datenrichtungs-Register B
1	1	–	–	Steuer-Register B

Bild 7-3: Adressieren der PIA-Register

Die internen Steuerregister

Jedes Steuerregister im 6520 legt, wie wir gesehen haben, in Bit 2 fest, auf welches der anderen Register für dieses Tor zugegriffen werden kann. Zusätzlich bietet es eine Reihe von Möglichkeiten für verschiedene Betriebsarten des Datentors an, darunter Erzeugen oder Übernehmen von Unterbrechungsanforderungen oder automatische Quittungsfunktionen. Eine vollständige Beschreibung dieser Möglichkeiten kann hier jedoch nicht gegeben werden. Man muß sich beim Einsatz des 6520 einfach nur auf das Datenblatt beziehen, das die Auswirkungen beim Setzen der verschiedenen Bits im Steuerregister angibt. Auf jeden Fall muß nach einer Rücksetzoperation jedes Steuerregister im 6520 mit den für den Anwendungszweck wichtigen Daten geladen werden.

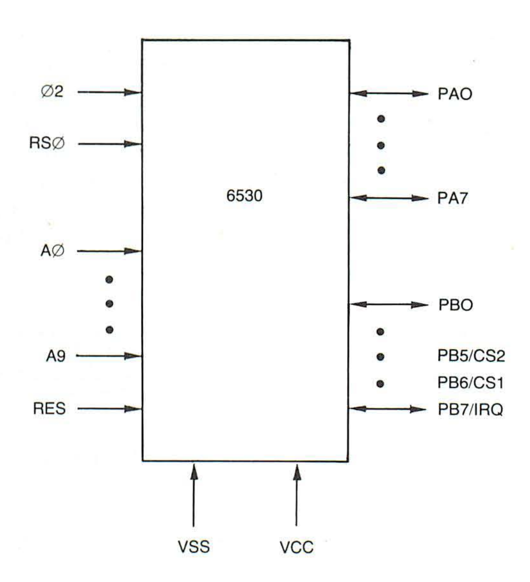


Bild 7-4: Anschlußbelegung des 6530-Bausteins

Der 6530

Der 6530 vereint vier Funktionen auf dem Chip: RAM, ROM, PIO und Zeitgeber. Das RAM bietet 64 x 8 Bits, das ROM 1 K x 8. Der Zeitgeber erlaubt verschiedene Möglichkeiten zur Intervallbestimmung, und der PIO-Teil entspricht im wesentlichen dem oben beschriebenen 6520: Es gibt 2 Tore, jedes mit Daten- und Datenrichtungsregister. Eine „0" im Datenrichtungsregister bestimmt eine Eingabe-, eine „1" eine Ausgabefunktion. Der programmierbare Zeitgeber kann bis 256 zählen (er enthält ein 8-Bit-Zählregister). Der Programmierer kann die Zählfrequenz als 1, 1/8, 1/64 oder 1/1024 vom Systemtakt vorgeben. Wenn die Vorgabe abgezählt ist, wird eine Unterbrechungsflagge auf „1" gesetzt. Der Inhalt des Zählregisters wird über den Datenbus bestimmt, die vier möglichen Zahlfrequenzen über Bits A0 und A1 im Adreßbus.
Drei Anschlüsse von Tor B spielen eine doppelte Rolle: PB5, PB6 und PB7 können für Steuerzwecke verwendet werden. Z. B. kann man PB7 zum Unterbrechungseingang bestimmen.
Der Chip wird insbesondere auf der KIM-1-Karte benutzt. (Beachten Sie: Beim KIM steht PB6 nicht zur freien Verfügung.)

Programmierung eines PIO

Betrachten wir als Beispiel ein Programm, das über Tor B eines 6520 oder eines 6522 den Wert 0 ausgeben soll. (Wir nehmen dabei an, daß das Steuerregister bereits gesetzt ist.)

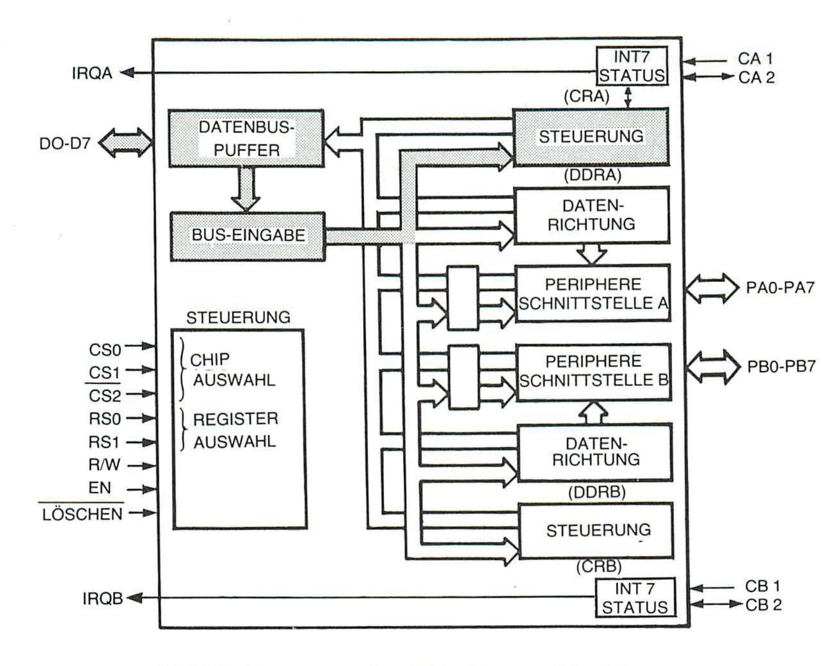


Bild 7-5: Anwendung eines PIA: Steuerregister laden

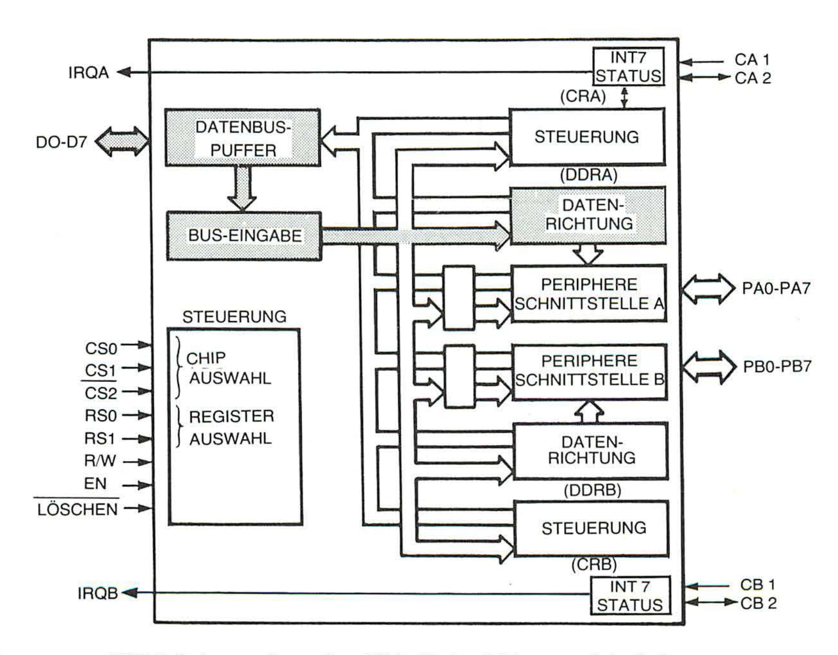


Bild 7-6: Anwendung eines PIA: Datenrichtungsregister laden

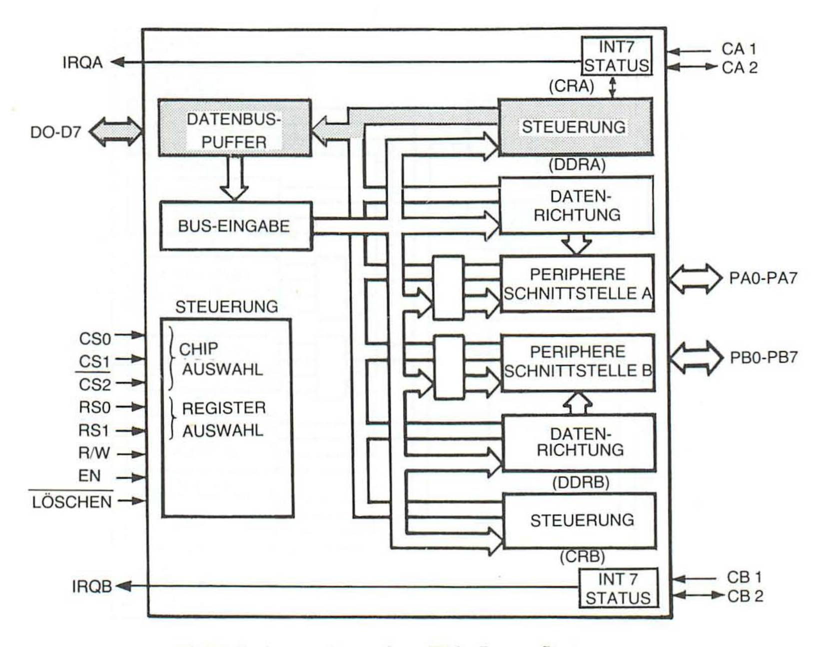


Bild 7-7: Anwendung eines PIA: Lesen Status

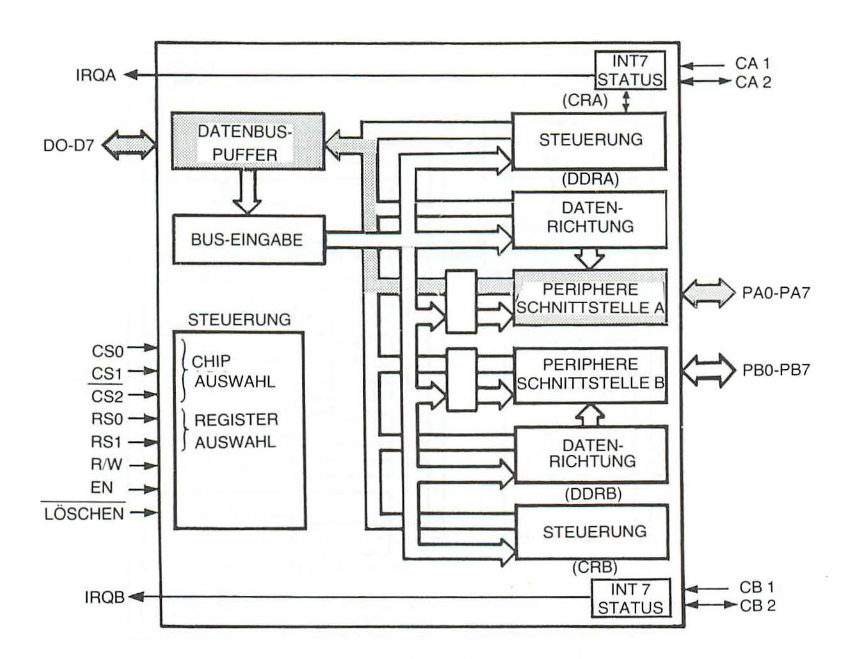


Bild 7-8: Anwendung eines PIA: Eingabe lesen

```
LDA  # $FF     ALLE LEITUNGEN AUF
STA  DDRB      AUSGABE SETZEN (TOR B)
LDA  CRB       STEUERREGISTER ABFRAGEN
ORA  # $04     ZUGRIFF AUF DATENREGISTER
STA  CRB       IN STEUERREGISTER SETZEN
LDA  # $00     DANN DEN WERT 0
STA  IORB      AUSGEBEN
```

DDRB ist die Adresse des Datenrichtungsregisters von Tor B, IORB ist prinzipiell dieselbe Adresse, nur ist hier das Datenregister gemeint. Zwischen den beiden Registerzugriffen muß erst das zugehörige Bit im Steuerregister umgeschaltet werden, was mit den Befehlen 3 bis 5 geschieht. Man kann den Inhalt des Steuerregisters auslesen, was eine einfache Handhabung bestimmter Bits ermöglicht. In unserem Beispiel setzen wir Bit 2 auf „1“. Man kann dann solange Daten über die Adresse IORB ausgeben, bis Bit 2 im Steuerregister von Tor B wieder auf „0“ gesetzt worden ist.

Der 6522

Der 6522-Baustein, auch „versatile interface adapter“ (VIA) genannt, was etwa „vielseitiger Interfaceadapter“ heißt, ist eine verbesserte Version des 6520-Bausteins. Zusätzlich zu dessen Möglichkeiten enthält der VIA zwei programmierbare Zeitgeber, einen seriell/parallel- und einen parallel/seriell-Umwandler sowie zugehörige Zwischenspeicher für die Daten. Die ausführliche Hardwarebeschreibung für diesen Baustein liegt außerhalb des Rahmens dieses Buchs hier. Mit der Beschreibung für die anderen Bausteine sollte es aber nicht schwierig sein, sich anhand der Datenblätter mit der Funktion der internen Register des 6522 als auch mit der Programmierung des Bausteins vertraut zu machen.

Der 6532

Der 6532-Chip ist ein Kombinationsbaustein, der ein 128 x 8-RAM, ein PIO mit zwei bidirektionalen Datentoren und einen programmierbaren Zeitgeber enthält. Er wird auf der von Synertek hergestellten SYM-Karte eingesetzt, die dem von MOS Technology und Rockwell gefertigten KIM-1 entspricht. Auch hier sei der Leser auf das Studium der Datenblätter verwiesen, um Funktion und Einsatz der verschiedenen internen Register kennenzulernen.

Zusammenfassung

Um von derartigen Bausteinen umfassend Gebrauch machen zu können, muß man leider die Bedeutung jedes einzelnen oder jeder zusammengehörigen Gruppe von Bits der verschiedenen Steuerregister kennen. Diese komplexen neuen Chips automatisieren eine Reihe von Aufgaben, die vorher durch Software oder besondere Logikschaltungen ausgeführt werden mußten. Insbesondere sind viele der Quittungsprozeduren bei Bausteinen wie dem 6522 automatisiert worden. Auch ist ein Teil der Unterbrechungssteuerung und -erkennung auf dem Chip verwirklicht worden. Mit der hier vermittelten Information sollte man in der Lage sein, die zugehörigen Daten-

blätter lesen und die Bedeutung der verschiedenen Register und Signale verstehen zu können. Natürlich werden ständig neue Bausteine auf den Markt kommen, die immer komplexere Algorithmen in Hardware umsetzen. Eine ausführliche Beschreibung der betreffenden E/A-Einheiten und -Techniken findet sich in dem Ergänzungsband D302D: „6502 Anwendungen".

KAPITEL 8
ANWENDUNGSBEISPIELE

Einführung

Dieses Kapitel soll dazu dienen, Ihre neuen Programmierfähigkeiten durch Besprechen einer Sammlung von Hilfsprogrammen zu schärfen. Diese Unterprogramme, auch „Routinen" genannt, finden sich häufig in der Praxis und werden gemeinhin als „Hilfsroutinen" bezeichnet. Um sie zu verstehen, müssen Sie mit den bis jetzt vermittelten Kenntnissen und Techniken vertraut sein.
Wir werden Zeichen von einer E/A-Einheit übernehmen und auf verschiedene Art und Weise bearbeiten. Zunächst aber wollen wir einen Bereich im Speicher löschen. (Das ist nicht unbedingt notwendig; jedes der vorgestellten Programme soll nur als Beispiel dienen.)

Löschen eines Speicherbereichs

Wir wollen den Inhalt des Speicherbereichs von Adresse BASIS bis Adresse BASIS + LÄNGE löschen (auf „0" setzen), wobei LÄNGE kleiner als 256 sein soll.

Das Programm lautet:

```
LOESCHEN LDX  # LAENGE  ZAEHLER SETZEN
         LDA  # 0       EINZUSCHREIBENDER WERT
NULLEN   STA  BASIS, X  EIN BYTE LOESCHEN
         DEX            ALLES ERFASST?
         BNE  NULLEN    NEIN: WEITERMACHEN
         RTS
```

Beachten Sie, daß Register X als Indexregister auf die gerade zu löschende Speicherstelle dient.

Der Akkumulator A wird einmal mit dem Wert „0" (binär 00000000) geladen und dann schrittweise in die gewünschten Speicherstellen (BASIS + LÄNGE, BASIS + LÄNGE 1, usw.) kopiert, bis X = 0 ist. Ist X auf Null heruntergezählt, so kehrt das Programm zurück. In einem Speichertest kann dieses Programm beispielsweise dazu dienen, den zu testenden Bereich zu löschen, um danach seinen Inhalt nachprüfen zu können.

Übung 8.1:
Schreiben Sie ein Speichertestprogramm, das zunächst einen 256-Byte-Block löscht und nachprüft, ob in allen Speicherstellen eine Null steht. Dann soll in alle Speicherstellen des Testblocks der Wert 01010101 geschrieben und anschließend der Inhalt auf Richtigkeit überprüft werden. Zum Abschluß ist in jedes Byte der Wert 10101010 zu schreiben und dann nachzuprüfen.

Als nächstes wollen wir E/A-Einheiten abfragen, ob eine von ihnen bedient werden muß.

Abfragen von E/A-Einheiten

Nehmen wir an, an unser System seien drei E/A-Einheiten angeschlossen. Ihre Statusregister befinden sich unter Adresse EASTAT1, EASTAT2 und EASTAT3. Wenn die Statusbits in Stelle 7 stehen, so lesen wir einfach die Statusregister und testen die Vorzeichenflagge N. Ansonsten können wir den BIT-Befehl des 6502 verwenden:

```
TEST    LDA  MASKE      TESTMASKE LADEN
        BIT  EASTAT1    STATUS 1 GESETZT?
        BNE  GERAET1    JA: GERAET 1 BEDIENEN
        BIT  EASTAT2    STATUS 2 GESETZT?
        BNE  GERAET2    JA: GERAET 2 BEDIENEN
        BIT  EASTAT3    SONST STATUS 3 GESETZT?
        BNE  GERAET3    JA: GERAET 3 BEDIENEN
```

Die MASKE kann z.B. den Wert 00100000 haben, wenn das Statusbit in Stelle 5 steht. Der BIT-Befehl setzt die Z-Flagge auf „1", wenn das Statusbit gelöscht ist und so die Verknüpfung MASKE UND EASTAT eine Null ergibt. Dies wird durch den BNE-Befehl getestet (branch if not equal zero – Verzweigen falls ungleich Null), der zur zugehörigen Bedienungsroutine springt, wenn das Statusbit gesetzt war.

Zeichen übernehmen

Nehmen wir an, wir hätten gerade festgestellt, daß von der Tastatur ein Zeichen übernommen werden kann. Wir wollen die von dort kommenden Zeichen in einem PUFFER genannten Speicherbereich ablegen, bis wir ein besonderes Zeichen namens SPEZ entdecken. Der Wert von SPEZ sei vorher im Programm festgelegt worden. Das Unterprogramm ZHOLEN übernimmt jeweils ein Zeichen von der Tastatur (nach den in Kapitel 6 beschriebenen Prinzipien) und liefert es im Akkumulator an. Wir nehmen an, daß höchstens 256 Zeichen übernommen werden, bevor SPEZ im Datenstrom auftaucht.

```
ZEICHEN    LDX #0         INDEX LOESCHEN (= 256 MAX.)
NAECHSTES JSR  ZHOLEN     ZEICHEN IN AKKU UEBERNEHMEN
           CMP #SPEZ      ABSCHLUSSZEICHEN?
           BEQ ALLES      JA: ALLES UEBERNOMMEN, ZURUECK
           STA PUFFER,X   SONST EINGABE IN PUFFER GEBEN
           INX            ZEIGER WEITERSETZEN
           JMP NAECHSTES  UND EINE NEUE EINGABE HOLEN
ALLES      RTS            ALLES UEBERNOMMEN: ZURUECK
```

Übung 8.2:
Dieses Grundprogramm läßt sich verbessern:
a) Schicken Sie das Zeichen an die Ausgabeeinheit zur Bestätigung zurück (z. B. an den Drucker einer Teletype).
b) Stellen Sie sicher, daß die Eingabe nicht länger als 256 Zeichen ist.

Damit haben wir eine Zeichenkette im Pufferspeicher stehen, die wir nun auf verschiedene Art bearbeiten wollen.

Ein Zeichen testen

Nehmen wir an, der weitere Programmablauf hänge davon ab, ob das Zeichen in Speicherstelle ZORT eine „0", eine „1" oder eine „2" ist. Ist es keines davon, soll eine Fehlermeldung durch die Routine NGEFUND ausgegeben werden:

```
ZTEST    LDA  ZORT      TESTZEICHEN HOLEN
         CMP  # $00     IST ES EINE „0"?
         BEQ  NULL      JA: ROUTINE NULL EINLEITEN
         CMP  # $01     IST ES EINE „1"?
         BEQ  EINS      JA: ROUTINE EINS EINLEITEN
         CMP  # $02     IST ES EINE „2"?
         BEQ  ZWEI      JA: ROUTINE ZWEI EINLEITEN
         JMP  NGEFUND   SONST FEHLER MELDEN
```

Wir lesen einfach ein Zeichen ein und testen es mit Hilfe der Vergleichsbefehle CMP.

Führen wir einen anderen Test durch:

Bereichstest

Versuchen wir herauszufinden, ob das in ZORT stehende Zeichen eine Ziffer zwischen „0" und „9" ist. Wenn Sie in der ASCII-Tabelle im Anhang nachschlagen, werden Sie finden, daß diese Ziffern aufeinanderfolgende Kodes, d.h. einen Kodebereich belegen. Wir müssen also testen, ob das Zeichen unter ZORT innerhalb dieser Grenzen liegt.
Das Testergebnis teilen wir dem übergeordneten Programm durch entsprechend gesetzte Flaggen mit: Ist C = 0, so handelt es sich um eine Ziffer; ist C = 1 und V = 0, so ist es eines der Zeichen mit kleinerem Kode; ist C = 1 und V = 1, so ist es eines der Zeichen mit größerem Kode.

```
ZIFFER    LDA  # $40     ZUNAECHST UEBERLAUFFLAGGE
          ADC  # $40     AUF 1 SETZEN (GROESSERFLAGGE)
          LDA  ZORT      BIT 7 AUF 1 SETZEN
          ORA  # $80     BIT 7 AUF 1 SETZEN
          CMP  # $B0     ASCII 0
          BCC  ZUKLEIN   UNTERSCHRITTEN: MELDEN
          CMP  # $B9     OBERGRENZE (ASCII 9)?
          BEQ  ISTZIFF   GENAU 9: ZIFFER MELDEN
          BCS  ZUGROSS   UEBERSCHRITTEN: MELDEN
ISTZIFF   CLC            UNGLEICHFLAGGE LOESCHEN
          RTS            UND ZURUECKSPRINGEN
ZUKLEIN   SEC            UNGLEICHFLAGGE SETZEN
          CLV            GROESSERFLAGGE LOESCHEN
ZUGROSS   RTS            UND SPRINGEN
```

Sehen Sie sich in Kapitel 4 noch einmal das Verhalten der verschiedenen Flaggen bei der Abarbeitung der Befehle ADC, ORA und CMP an. CMP beeinflußt die Flaggen N, Z und C, ORA die Flaggen N und Z. *Weder CMP noch ORA beeinflussen die Überlaufflagge V.* Das heißt, wenn wir V zu Anfang auf „1" setzen, so behält es seinen Wert durch ORA und CMP hindurch. Da uns V als „größer"-Meldung dient, müssen wir es irgendwo auf „1" setzen, am besten vor den Test, um die Übersicht zu wahren. Leider besitzt der 6502 keinen Befehl, mit dem V unmittelbar gesetzt werden kann. Wir erreichen dasselbe durch die beiden ersten Befehle.
Der Vergleichsbefehl CMP setzt die C-Flagge immer dann auf Eins, wenn der Wert im Akkumulator größer oder gleich dem Vergleichswert ist. Hat der Akkumulator einen kleineren Inhalt, so wird C = 0 gesetzt.
Ist unser Zeichen keine Ziffer, so ist es entweder kleiner als ASCII-„0" (B0, wenn das Paritätsbit auf „1" gesetzt ist) oder größer als ASCII-„9" (B9). Ist es im ersten Vergleich kleiner, so ist die Verzweigungsbedingung nach ZUKLEIN erfüllt, C hat den Wert 0, V hat von oben den Wert 1. Beides muß von ZUKLEIN vor dem Rücksprung gelöscht werden. Ist das Testzeichen eine „9", so ist im zweiten Vergleich Z = 1 und C = 1.
Da „9" eine Ziffer ist, müssen wir vor dem Rücksprung erst die Fehlerflagge C löschen. Der Wert von V ist dann unwesentlich. Ist dagegen der Akkumulatorinhalt größer als „9", so erhalten wir C = 0, V = 1 und können unmittelbar zurückkehren (über Verzweigung nach ZUGROSS). Ansonsten ist es eine Ziffer, C = 0. Wir bräuchten C nicht mehr zu löschen; da es aber wegen des Falls „9" ohnehin notwendig ist, machen wir aus Ersparnisgründen einfach damit weiter.
Man könnte natürlich auch andere Formen der Rückmeldung an das Hauptprogramm einschlagen, wie z.B. Löschen des Akkumulators, falls keine Ziffer vorliegt.

Übung 8.3:
Vereinfachen Sie das obenstehende Programm, indem Sie die Obergrenze auf das auf „9" folgende ASCII-Zeichen legen.

Übung 8.4:
Untersuchen Sie, ob das Zeichen unter ZORT ein Großbuchstabe ist. Geben Sie die gleichen Flaggen zurück wie im numerischen Fall.

Wir haben in unserem Beispiel das Paritätsbit von vornherein auf „1" gesetzt. Das ist eine reine Vorsichtsmaßnahme (wir hätten auch „0" nehmen können). Wir können nicht unbedingt garantieren, daß die Eingabeeinheit keine Paritätsinformation aussendet. Die einfache ASCII-Form von „0" lautet z.B. 0110000, umfaßt also nur 7 Bits. Benutzen wir gerade Parität, so ergibt das Byte 00110000, bei ungerader Parität aber 10110000. Es leuchtet ein, daß wir mit unserem Test in große Schwierigkeiten kämen, müßten wir die Paritätsinformation mit berücksichtigen. In einigen Fällen jedoch benötigt ein Ausgabegerät eben solche Paritätsinformation. Sehen wir uns also an, wie man diese ermittelt.

Paritätserzeugung

Dieses Programm erzeugt gerade Parität in Bit 7.

```
PARITAET  LDX # $07         7 BITS ZU ZAEHLEN
          LDA # $00         INITIALISIEREN DES
          STA EINSBIT       EINS-BIT-ZAEHLERS
          LDA ZPOS          ZEICHEN HOLEN
          ROL A             BIT 7 AUSBLENDEN
NAECHSTES ROL A             NAECHSTES BIT = 1?
          BCC NULL          NEIN: NICHT ZAEHLEN
EINS      INC EINSBIT       SONST EINSEN ZAEHLEN
NULL      DEX               ALLE BITS ERFASST?
          BNE NAECHSTES     NEIN: WEITERTESTEN
          ROL A             BIT 0 ZURUECKHOLEN
          ROL A             BIT 7 IN C SCHIEBEN
          LSR EINSBIT       RECHTES ZAEHLERBIT = PARITAET
          ROR A             UEBER C IN A EINSCHIEBEN
          RTS               UND ZURUECKSPRINGEN
```

Mit Register X zählen wir die nach links durch den Akkumulator rotierten Bits. Jedesmal, wenn eine „1" in das Übertragsbit C geschoben worden ist, wird (getestet durch den BCC-Befehl) der Einserzähler EINSBIT weitergezählt. Sind acht Bits verschoben worden (das Programm vernachlässigt Bit 7, das die Paritätsinformation erhalten soll), so wird A noch weitere zwei Stellen durch C rotiert. Dadurch kommt der Platz des Paritätsbits wieder in C.
Der richtige Paritätswert steht im niederstwertigen Bit des Einserzählers EINSBIT. Dieses Bit wird durch einen LSR-Befehl in C geschoben und gelangt von dort mittels ROR in Stelle 7 von A.

Übung 8.5:
Prüfen Sie nach, ob das Programm tatsächlich die richtige Paritätsinformation erzeugt. Berechnen Sie erst die Parität mit dem Programm und vergleichen Sie das Ergebnis mit der erwarteten Information.

Kodeumwandlung: ASCII in BCD

ASCII-Ziffern lassen sich sehr einfach in die zugehörige BCD-Darstellung umwandeln. Die hexadezimale Form der ASCII-Ziffern „0“ bis „9“ lautet B0 bis B9 mit und 30 bis 39 ohne gesetztes Paritätsbit. Die BCD-Darstellung ergibt sich ganz einfach durch Ausblenden des „B“, d.h. durch Ausmaskieren des linken Nibble (der vier linken Bits):

```
ASCBCD    LDA  ZORT      ZEICHEN HOLEN
          AND  # $0F     LINKES NIBBLE AUSBLENDEN
          STA  BCDORT    UND BCD-ZEICHEN ABLEGEN
```

Übung 8.6:
Schreiben Sie ein Programm BCDASC, das BCD-Zeichen in ASCII-Form bringt.

Übung 8.7:
(etwas schwieriger) Schreiben Sie ein Programm, das eine BCD-Zahl in binäre Darstellung bringt.

Hinweis:
Die BCD-Zahl $N_3 N_2 N_1 N_0$ hat den Wert $(((N_3 \times 10) \times N_2) \times 10 + N_1) \times 10 + N_0$.
Um mit 10 zu multiplizieren, schieben Sie die Zahl nach links (x 2), nochmal nach links (x 4), addieren den ursprünglichen Wert (x 5) und schieben schließlich noch das Ergebnis eine Stelle nach links (x 10).
In voller BCD-Notation kann das erste Wort die Zahl der verwendeten BCD-Ziffern, das nächste Nibble die Stelle des Kommas und alle folgenden die eigentlichen BCD-Ziffern enthalten. (Nehmen wir an, es sei kein Komma vorhanden.) Das letzte Nibble im Block kann u.U. unbenutzt sein.

Das größte Element einer Tabelle aufsuchen

Auf Seite Null möge in Speicherstelle BASIS die Anfangsadresse einer Tabelle stehen. Der erste Eintrag in die Tabelle soll ihre Länge angeben. Das folgende Programm sucht dann nach dem größten Tabellenelement. Sein Wert wird in A zurückgeliefert, sein Ort in der Tabelle in Speicherstelle INDEX.
Das Programm benutzt die Register A und Y und setzt indirekte Adressierung ein, so daß man beliebig im Speicher stehende Tabellen durchsuchen kann.

```
MAX       LDY  # 0          TABELLENINDEX INITIALISIEREN
          LDA  (BASIS), Y   ERSTEN EINTRAG HOLEN (LAENGE)
          TAY               UND ALS INDEX VERWENDEN
          LDA  # 0          MAXIMALWERT AUF NULL SETZEN
          STA  INDEX        MAXIMALORT = TABELLENANFANG
SUCHEN    CMP  (BASIS), Y   A KLEINER ALS TABELLENELEMENT?
          BCS  NAENDERN     NEIN: NICHTS NEU SETZEN
          LDA  (BASIS), Y   SONST MAXIMALWERT NEU SETZEN
          STY  INDEX        UND MAXIMALSTELLE VERMERKEN
NAENDERN  DEY               ALLES ERFASST?
          BNE  SUCHEN       NEIN: WEITERTESTEN
          RTS               SONST ZURUECKSPRINGEN
```

Das Programm untersucht zuerst die N-te Tabellenstelle und setzt A auf deren Wert, falls sie größer als Null ist. Zugleich wird der Tabellenort in INDEX festgehalten. Dann folgt ein Test des (N-1)ten Tabellenelements usw.
Dieses Programm arbeitet mit positiven ganzen Zahlen.

Übung 8.8:
Schreiben Sie das Programm so um, daß positive und negative Zweierkomplementzahlen bearbeitet werden können.

Übung 8.9:
Läßt sich das Programm auch für ASCII-Zeichen verwenden?

Übung 8.10:
Schreiben Sie ein Programm, das N Zahlen in aufsteigender Reihenfolge sortiert.

Übung 8.11:
Schreiben Sie ein Programm, das N Wörter (zu je 3 Zeichen) in alphabetische Reihenfolge bringt.

Summe über N Elemente

Dieses Programm berechnet die 16-Bit-Summe von N Tabelleneinträgen. Die Anfangsadresse der Tabelle steht in Speicherstelle BASIS auf Seite Null. Der erste Tabelleneintrag enthält die Anzahl N an Tabellenelementen. Die 16-Bit-Summe wird in den beiden Speicherstellen SUMNW (niederwertige Hälfte) und SUMHW (höherwertige Hälfte) abgelegt. Die Summe umfaßt auch bei längeren Ergebnissen nur die 16 niederwertigen Bits. (Man sagt, die höherwertigen Bits sind „abgebrochen“ [truncated] worden).
Das Programm verändert Register A und Y. Es wird eine maximale Tabellenlänge von 256 Elementen angenommen.

```
TABSUM    LDA  #0          SUMME INITIALISIEREN
          STA  SUMNW       NIEDERWERTIGE HAELFTE
          STA  SUMHW       HOEHERWERTIGE HAELFTE
          TAY              ERSTES ELEMENT BEZEICHNEN
          LDA  (BASIS),Y   UND TABELLENLAENGE
          TAY              ALS INDEXZEIGER BENUTZEN
          CLC              ADDITION VORBEREITEN
ADDIEREN  LDA  (BASIS),Y   NAECHSTES ELEMENT HOLEN
          ADC  SUMNW       UND AUFADDIEREN
          STA  SUMNW       NIEDERWERTIGEN TEIL SPEICHERN
          BCC  CNULL       KEIN UEBERTRAG: VERZWEIGEN
          INC  SUMHW       SONST UEBERTRAG AUFRECHNEN
          CLC              NAECHSTE ADDITION VORBEREITEN
CNULL     DEY              ALLES ERFASST?
          BNE  ADDIEREN    NEIN: NAECHSTES ELEMENT
          RTS              SONST ZURUECKSPRINGEN
```

Übung 8.12:
Ändern Sie das Programm so ab, daß
a) eine 24-Bit-Summe,
b) eine 32-Bit-Summe berechnet und
c) jeder Überlauf angezeigt wird.

Eine Prüfsummenberechnung

Eine Prüfsumme ist eine Ziffer oder eine Zahl, die aus einem zusammenhängenden Block von Zeichen berechnet wurde. Man ermittelt die Prüfsumme beim Speichern der Daten und setzt sie in der Regel an das Ende des dazugehörigen Datenblocks. Um die Richtigkeit der Daten zu überprüfen, berechnet man die Prüfsumme nach demselben Verfahren neu und vergleicht das Ergebnis mit dem gespeicherten Wert. Jede Abweichung der beiden Prüfsummen gibt dann an, daß ein Fehler vorliegen muß.
Man benutzt dabei verschiedene Algorithmen. Wir wollen hier alle Bytes einer Tabelle von N Elementen miteinander EXKLUSIV-ODER-verknüpfen und das Ergebnis im Akkumulator zurückgeben. Wie bisher beginnt die Tabelle an der in Seite Null unter BASIS abgelegten Speicheradresse und enthält die Zahl N der Tabellenelemente als ersten Eintrag. Das Programm verändert A und Y; N muß kleiner als 256 sein.

```
PRUEFSUMME LDY  # 0         ERSTEN EINTRAG
           LDA  (BASIS), Y  ALS TABELLENZEIGER
           TAY              IN Y UEBERNEHMEN
           LDA  # 0         PRUEFSUMME INITIALISIEREN
PRSCHLEIFE EOR  (BASIS), Y  VERKNUEPFEN DER EINTRAEGE
           DEY              ALLES ERFASST?
           BNE  PRSCHLEIFE  NEIN: NAECHSTES ELEMENT
           RTS              SONST ZURUECK
```

Nullen zählen

Das folgende Programm zählt die Nullbytes in unserer Standardtabelle und gibt sie in Register X zurück. Dabei werden A, X und Y verändert.

```
BYTEZAEHLER LDY  # 0         ERRSTEN EINTRAG
            LDA  (BASIS), Y  ALS TABELLENZEIGER
            TAY              IN Y UEBERNEHMEN
            LDX  # 0         BYTEZAEHLER INITIALISIEREN
ZAEHLEN     LDA  (BASIS), Y  IST DER NAECHSTE EINTRAG = 0?
            BNE  NICHTZLN    NEIN: NICHT ZAEHLEN
            INX              SONST IN X AUFZAEHLEN
NICHTZLN    DEY              ALLE BYTES ERFASST?
            BNE  ZAEHLEN     NEIN: WEITERTESTEN
            RTS              SONST RUECKSPRUNG
```

Übung 8.13:
Verändern Sie dieses Programm zum Zählen
a) aller Sterne (★),
b) aller Großbuchstaben
c) aller Ziffern zwischen 0 und 9.

Zeichenkette suchen

Im Speicher sei, wie in Bild 8-1 dargestellt, eine Kette (string) von Zeichen (z. B. ein Briefabschnitt) festgehalten. Wir wollen diese Zeichenkette nach einer bestimmten anderen (etwa einem Namen) durchsuchen. Diese „Schablone" (template) sei unter TEMPLT abgelegt, ihre Länge möge in TPTLEN stehen. Die Länge der Originalkette stehe in STRLEN. Das Programm liefert in Register X entweder die Stelle der gesuchten Kette oder – falls nicht zu finden – hexadezimal FF zurück. Bild 8-2 zeigt das Flußdiagramm für das Problem. Zunächst wird nach dem ersten Zeichen von TEMPLT gesucht. Taucht dieses Zeichen in der Originalkette nirgends auf, so endet das Programm mit negativem Ergebnis. Wird das Zeichen irgendwo gefunden, so muß geprüft werden, ob die folgenden Zeichen übereinstimmen. Falls nicht, wird die Suche nach dem ersten Zeichen fortgesetzt, da die gesuchte Kette ja auch weiter hinten im Original stehen kann. Das zugehörige Programm zeigt Bild 8-3. Beachten Sie, daß Register X während der Suche als Zeiger auf das jeweils im Original untersuchte Zeichen dient. Man setzt hier selbstverständlich indizierte Adressierung ein.

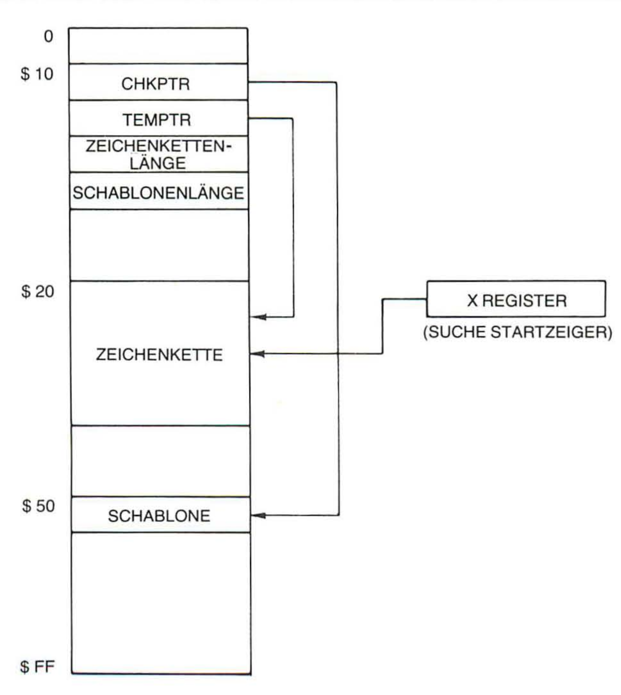


Bild 8-1: Durchsuchen einer Zeichenkette: Register- und Speicherbelegung

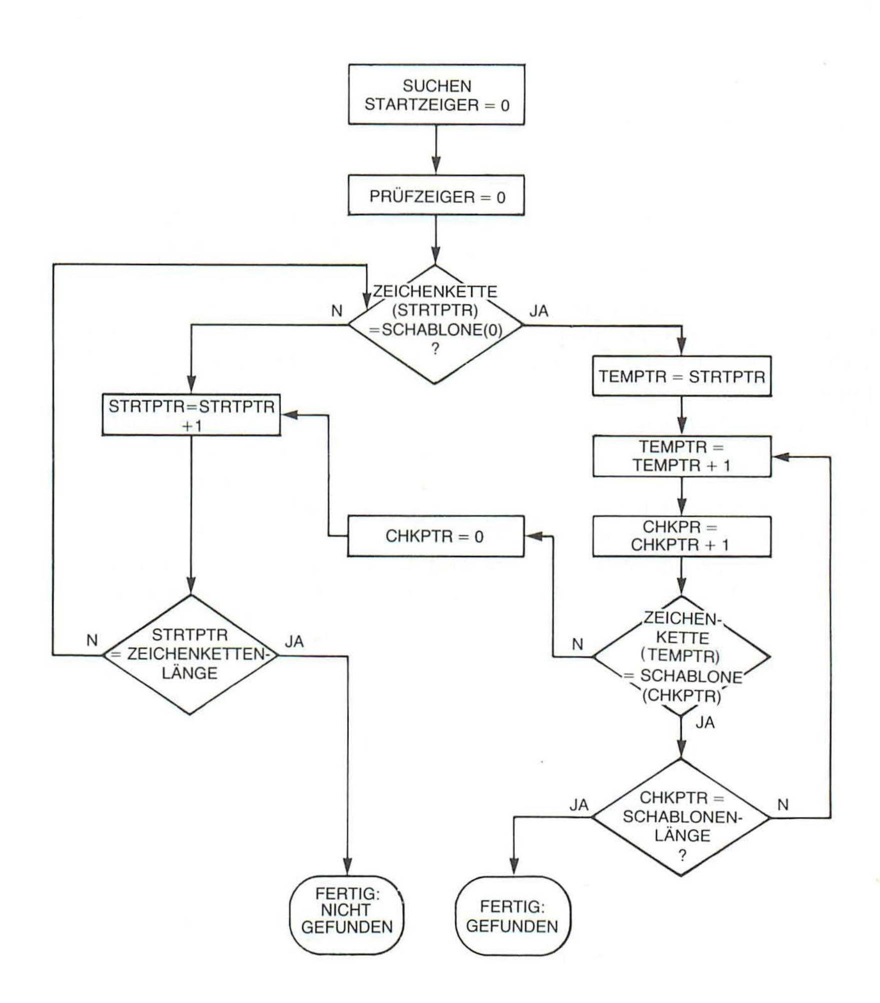


Bild 8-2: Durchsuchen einer Zeichenkette: Flußdiagramm

```
LOC    CODE       LINE

                  ; SUCHE ERSTES AUFTRETEN DER ZEICHENFOLGE 'TEMPLT'
                  ; IN DER ZEICHENKETTE 'STRING'.
                  ;
0000              STRING   =  $20         ; ANFANGSADRESSE FUER STRING
0000              TEMPLT   =  $50         ; ANFANGSADRESSE FUER TEMPLT
                  ;
0000                       * = $10
                  ;
0010   00         CHKPTR   * = * + 1
0011   00         TEMPTR   * = * + 1
0012   00         STRLEN   * = * + 1      ; LAENGE DES STRING
0013   00         TPTLEN   * = * + 1      ; LAENGE DES TEMPLT
0014                       * = $200
0200   A2 00               LDX #0         ; INITIALISIERUNG DES SUCHZEIGERS.
0202   A5 50      NXTPOS   LDA TEMPLT     ; ERSTES TEMPLT-ELEMENT ...
0204   D5 20               CMP STRING,X   ; = AKTUELLEM STRING-ELEMENT ?
0206   F0 08               BEQ CHECK      ; FALLS JA, UEBERPRUEFE DEN REST ...
0208   E8         NXTSTR   INX            ; NEIN, SUCHZEIGER WEITERSETZEN;
0209   E4 12               CPX STRLEN     ; IST DIESER GLEICH STRLEN ?
020B   D0 F5               BNE NXTPOS     ; NEIN, PRUEFE NAECHSTES STRING-ELEMENT.
020D   A2 FF               LDX #$FF       ; JA, SETZE ANZEIGE ' TEMPLT NICHT GEFUNDEN '
020F   60                  RTS            ; ALLE ELEMENTE GETESTET, RUECKKEHR ZUM AUFRUFER.
0210   86 11      CHECK    STX TEMPTR     ; SETZE ARBEITSZEIGER AUF SUCHZEIGER.
0212   A9 00               LDA #0
0214   85 10               STA CHKPTR     ; INITIALISIERE TEMPLT-ZEIGER AUF ERSTES ELEMENT
0216   E6 11      CHKLP    INC TEMPTR     ; SETZE ARBEITSZEIGER AUF NAECHSTES STRING-ELEMENT
0218   E6 10               INC CHKPTR     ; SETZE TEMPLT-ZEIGER AUF NAECHSTES ELEMENT
021A   A4 10               LDY CHKPTR
021C   C4 13               CPY TPTLEN     ; IST DIESER GLEICH TPTLEN ?
021E   F0 0C               BEQ FOUND      ; FALLS JA, WURDE TEMPLT IN STRING GEFUNDEN.
0220   B9 50 00            LDA TEMPLT,Y   ; NEIN, HOLE AKTUELLES TEMPLT-ELEMENT...
0223   A4 11               LDY TEMPTR
0225   D9 20 00            CMP STRING,Y   ; IST ES GLEICH DEM AKTUELLEN STRING-ELEMENT ?
0228   D0 DE               BNE NXTSTR     ; FALLS NEIN, SETZE SUCHE NACH ERSTEM TEMPLT-ELEMENT
022A                                      ;  IN STRING FORT.
022A   F0 EA               BEQ CHKLP      ; JA, SUCHE WEITERE UEBEREINSTIMMUNGEN.
022C   60         FOUND    RTS            ; FERTIG, RUECKKEHR ZUM PUNKT DES AUFRUFS.
022D                       .END
```

Bild 8-3: Durchsuchen einer Zeichenkette: Das Programm

Zusammenfassung

Wir haben in diesem Kapitel allgemein brauchbare Hilfsprogramme vorgestellt, welche die in den vorigen Kapiteln eingeführten Techniken in der Praxis anwenden. Diese Beispiele sollten Sie in die Lage versetzen, Ihre eigenen Probleme lösen zu können. Die meisten der hier vorgestellten Programme haben eine besondere Datenstruktur benutzt: die Tabelle. Es gibt jedoch einige Möglichkeiten mehr, Daten zu strukturieren. Diese wollen wir als nächstes untersuchen.

KAPITEL 9
DATENSTRUKTUREN
Teil 1: ENTWURFSPRINZIPIEN

Einführung

Das Erstellen guter Programme beinhaltet zwei Aufgaben: *Algorithmusentwicklung und Entwurf von Datenstrukturen.* Die meisten einfachen Programme stellen keine besonderen Ansprüche an die verwendeten Datenstrukturen, so daß hier nur das Problem gelöst werden muß, ein Programm zu entwerfen und es optimal in einer gegebenen Maschinensprache zu kodieren. Das haben wir bis jetzt getan. Entwicklung komplexerer Programme erfordert jedoch ein grundlegendes Verständnis der anzuwendenden Datenstrukturen. Wir haben bis jetzt zwei dieser Strukturen verwendet: Tabelle und Stapel. In diesem Kapitel hier sollen weitere, allgemeinere Datenstrukturen vorgestellt werden, die zur Problemlösung nützlich sein können. Dieses Kapitel hier ist vollständig unabhängig vom Mikroprozessor oder Computer, den man einsetzt. Es behandelt auf theoretischer Ebene die logische Datenorganisation im System. Zum Thema Datenstrukturen gibt es ganze Bücher, ebenso zur Frage leistungsfähiger Multiplikations-, Divisions- und anderer allgemeiner Algorithmen. In einem einzigen Kapitel kann daher nur ein allgemeiner Überblick gegeben werden, der notwendigerweise auf die allerwichtigsten Grundlagen beschränkt bleiben muß. Es kann und soll hier keinerlei Vollständigkeit angestrebt werden.
Sehen wir uns nun die verbreitetsten Datenstrukturen an:

Zeiger

Ein Zeiger (pointer) ist eine Zahl, die den Ort der betrachteten Daten angibt. Jeder Zeiger ist eine Adresse. Jedoch ist bei weitem nicht jede Adresse auch ein Zeiger. Man kann eine Adreßangabe nur dann als Zeiger betrachten, wenn sie bestimmte Daten oder sonstige Informationsgruppen bezeichnet. Mit dem Stapelzeiger haben wir bereits ein typisches Exemplar eines Zeigers kennengelernt. Er zeigt auf die Spitze des zugeordneten Stapels (in den meisten Fällen eigentlich auf die erste freie Stelle darüber). Wir hatten den Stapel als eine besondere Datenstruktur, als LIFO-Struktur (last-in, first-out) eingeführt, was wir gleich noch einmal im Zusammenhang betrachten wollen.

Ein anderes Beispiel für Einsatz von Zeigern trat bei der indirekten Adressierung auf. Die indirekte Adresse ist immer ein Zeiger, der auf die gewünschten Daten deutet.

Übung 9.1:
Betrachten Sie Bild 9-1. Unter Adresse 15 findet sich im Speicher ein Zeiger, der die Tabelle T bezeichnet. Tabelle T beginnt bei Adresse 500. Wie lautet dann der Inhalt des Zeigers auf T?

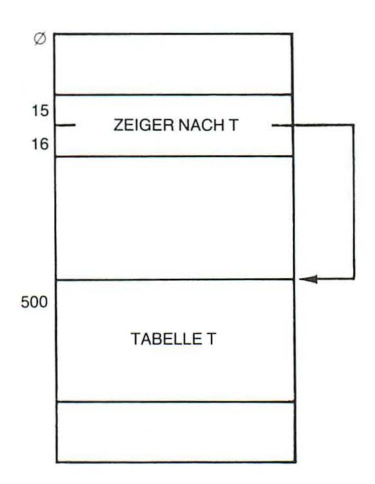


Bild 9-1: Indirekter Zugriff durch einen Zeiger

Listen

Nahezu alle Datenstrukturen sind in Form verschiedenster Listen organisiert.

Sequentielle Listen

Eine sequentielle Liste, in Form einer Tabelle oder eines Blocks, ist wahrscheinlich die einfachste Datenstruktur. Wir haben sie die ganze Zeit hindurch benutzt. Üblicherweise sind Tabellen nach bestimmten Gesichtspunkten geordnet, z.B. in alphabetischer Reihenfolge oder nach der Größe von Zahlen. Man kann dann, etwa mit Hilfe indizierter Adressierung, einfach auf bestimmte Tabellenelemente zugreifen. Ein Block bezieht sich dagegen in der Regel auf eine Datengruppe, die zwar innerhalb bestimmter Grenzen steht, deren Inhalt aber nicht geordnet ist.
Ein Block kann z.B. Zeichenketten (etwa einen Text) enthalten. Es kann sich um einen Sektor einer Diskettenaufzeichnung handeln. Oder es mag ein bestimmter logischer Abschnitt (ein Segment) im Speicher sein. In all diesen Fällen kann es Schwierigkeiten bereiten, will man auf bestimmte Datenelemente im Block zugreifen.
Um die in Blöcken festgehaltene Information wiederzugewinnen, werden oft Verzeichnisse benutzt.

Verzeichnisse

Ein Verzeichnis (directory) ist eine Liste über Tabellen oder Blöcke. So verwendet man z.B. in einer Datei normalerweise Verzeichnisse, um auf die Information zuzugreifen. Ein einfaches Beispiel: Das Hauptverzeichnis einer Datenbank möge die Namen aller ihrer Benutzer umfassen. Das geschieht, wie in Bild 9-2 dargestellt, in Form einer Liste. Der Eintrag für einen bestimmten Benutzer, sagen wir für „Hans", zeigt dann auf das für ihn zuständige Unterverzeichnis. Dieses Unterverzeichnis ist eine Liste aller für oder von Hans angelegter Dateien (files) und deren Ort in der Datenbank. Das ist wiederum eine Zeigertabelle. Wir haben damit ein in zwei Ebenen aufgestelltes Verzeichnis erhalten. Ein flexibles Verzeichnissystem wird noch andere Zwischenverzeichnisse enthalten, wie sie der jeweilige Benutzer gerade als sinnvoll empfindet.

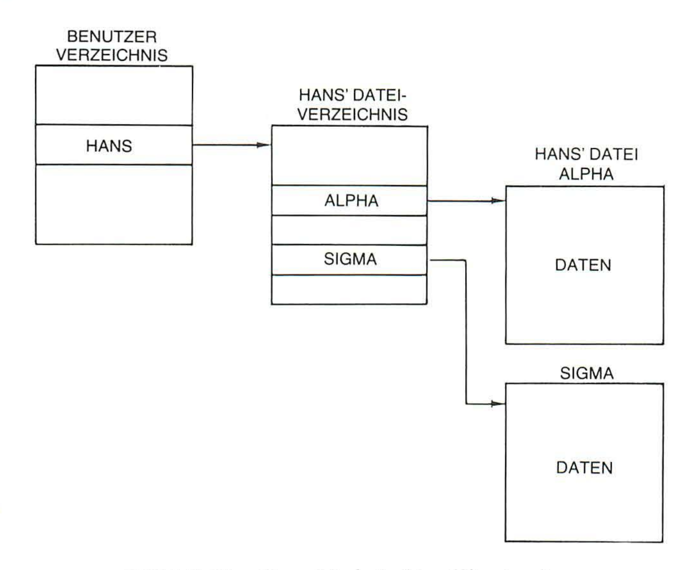


Bild 9-2: Eine Verzeichnisstruktur (directory)

Verkettete Listen

Es gibt in einem System oft Informationsblöcke (Daten, Ereignisse o. ä.), die nur schwierig verschoben werden können. Wären sie einfach zu verschieben, so könnte man sie zum Sortieren oder Strukturieren in Tabellen zusammenfassen. Wir wollen sie aber lieber dort stehen lassen wo sie sind und doch irgendwie eine Ordnung hineinbringen, wie z.B. das erste Element, das zweite Element usw. Dieses Problem läßt sich durch eine verkettete Liste (linked list) lösen, ein in Bild 9-3 verdeutlichtes Konzept. Dort zeigt ein Listenzeiger namens FIRSTBLOCK auf den Anfang des ersten Blocks. Innerhalb dieses Blocks befindet sich, etwa als erstes oder letztes Wort, eine besondere Speicherstelle, die einen Zeiger PTR1 auf den zweiten Block enthält. Block 2 enthält entsprechend einen Zeiger auf den dritten Block. Da dieser der letzte

Eintrag in unserer Datei ist, trägt der in ihm enthaltene Zeiger eine spezielle Endinformation, zumeist NIL (nichts) genannt, oder er zeigt einfach auf sich selbst zurück, so daß das Ende der Liste erkennbar ist. Diese Struktur ist recht ökonomisch, da sie nur ein paar Zeiger benötigt (einen pro Block) und den Benutzer von der Aufgabe befreit, die Blöcke physisch im Speicher verschieben zu müssen.

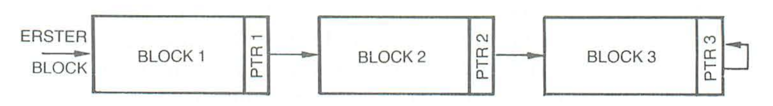


Bild 9-3: Eine verkettete Liste

Betrachten wir als Beispiel, wie ein neuer Block in die Liste eingefügt werden kann (Bild 9-4). Nehmen wir an, der neue Block trage die Adresse NEWBLOCK und sollte zwischen Block 1 und Block 2 eingeschoben werden. Dazu wird einfach der Zeiger PTR1 in Block 1 auf NEWBLOCK gesetzt (d.h. er bekommt diese Adresse als neuen Inhalt), so daß er jetzt auf Block X zeigt. Der dort befindliche Zeiger PTRX wird auf Block 2 gesetzt, d. h. erhält den vorigen Wert von PTR1. Die anderen Zeiger bleiben unverändert. Damit erfordert das Einfügen eines Blocks lediglich das Ändern zweier Zweige in der Struktur, was diese sehr leistungsfähig macht.

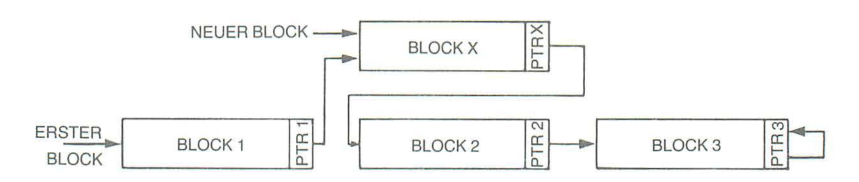


Bild 9-4: Eine verkettete Liste: Einfügen eines neuen Blocks

Übung 9.2:
Zeichnen Sie auf, wie man aus dieser Struktur Block 2 entfernen kann.

Für besondere Zugriffserfordernisse und für besondere Anforderungen an die einzufügenden bzw. zu löschenden Listenelemente hat man verschiedene Grundformen von Listenstrukturen entwickelt, von denen wir die meistgebrauchten verketteten Listen untersuchen wollen.

Warteschlangen

Eine Warteschlange (queue) wird formal FIFO-Liste (first-in, first-out) genannt. Sie ist in Bild 9-5 dargestellt. Nehmen wir dort z.B. an, der links wiedergegebene Kasten sei eine Druckerroutine. Die rechts stehenden Kästen mögen Druckanforderungen von verschiedenen anderen Programmteilen her darstellen. Sie werden dann in der Reihenfolge abgearbeitet, in der sie sich in die Warteschlange eingereiht haben. Als nächstes kommt in der Zeichnung Block 1 an die Reihe, dann Block 2 und schließlich Block 3.

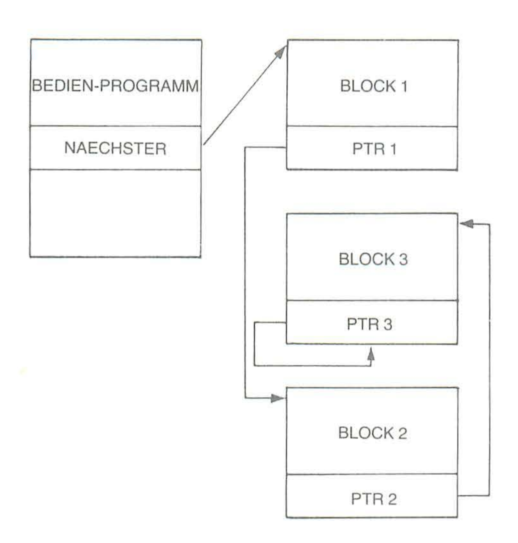


Bild 9-5: Eine Warteschlange (FIFO-Struktur)

Neue Daten werden immer am Ende einer Warteschlange eingereiht. Hier würde ein neuer Anforderungsblock nach PTR3 angefügt werden. Das garantiert, daß der erste in die Schlange eingefügte Block auch als erster bearbeitet wird. Warteschlangen sind in Computersystemen recht verbreitet, wo bestimmte Daten und Ereignisse auf ihre Abarbeitung, beispielsweise durch einen Prozessor oder langsame Peripherieeinheiten warten müssen.

Stapel

Die Stapelstruktur haben wir durch das ganze Buch hindurch benutzt. Es handelt sich um eine LIFO-Struktur (last-in, first-out), bei der das zuletzt (auf der Stapelspitze) abgelegte Element als erstes wieder entnommen wird. Ein Stapel kann als sortierter Block oder als verkettete Liste aufgebaut werden. Da bei Mikroprozessoren ein Stapel in der Regel für schnelle Ereignisse, wie Unterprogramm- oder Unterbrechungsverwaltung, eingesetzt wird, stellt man in der Regel einen zusammenhängenden Block im Speicher ab, anstatt eine verkettete Liste zu benutzen.

Block oder verkettete Liste?

Ganz entsprechend könnte man für eine Warteschlange einen zusammenhängenden Speicherblock reservieren. Der Vorteil eines solchen Speicherblocks ist der rasche Zugriff auf die Information und der Wegfall der Zeiger in der Information. Nachteilig ist dagegen die in der Regel umfangreiche Blockgröße, die man für den ungünstigsten Fall reservieren muß. Es ist außerdem schwierig, in einen Block Elemente hineinzubringen bzw. von dort herauszunehmen. Da es traditionell bei Mikrocomputern an Speicherraum mangelt, werden Speicherblöcke in der Regel für Strukturen mit konstanter Größe oder für solche mit raschem Informationszugriff reserviert.

Geschlossene Listen

Eine geschlossene Liste (circular list) ist eine Ringstruktur, eine verkettete Liste, bei der der letzte Zeiger wieder auf den Anfang zurückweist, wie es Bild 9-6 zeigt. Man verwaltet in so einem Fall zumeist einen besonderen Zustandszeiger (current-event pointer), der die gerade bearbeitete Stelle bezeichnet. Wenn neue Ereignisse eintreten oder Programme auf Abarbeitung warten, dann wird dieser Zustandszeiger eine Stelle nach rechts oder links bewegt. Eine geschlossene Liste wird in der Regel dort eingesetzt, wo alle Blöcke gleiche Priorität besitzen. Geschlossene Listen werden aber auch gerne als Untergliederung anderer Strukturen bei Suchoperationen benutzt, weil man sehr rasch vom Ende auf den Anfang zurückgreifen kann.

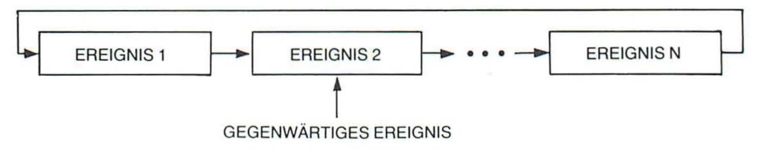


Bild 9-6: Eine geschlossene Liste

Ein Anwendungsfall einer geschlossenen Liste ist ein Abfrageprogramm, das üblicherweise immer rundum läuft, alle Einheiten abfragt, bei Bedarf bedient, und das nach Bearbeiten der letzten Einheit wieder zur Abfrage der ersten übergeht. Das Beispiel zeigt auch, daß diese Strukturen verketteter Listen nicht auf Daten beschränkt sind, sondern zur Strukturierung der Programmabarbeitung selbst dienen können.

Bäume

Wenn alle Elemente einer Struktur untereinander in logischer Beziehung stehen (man nennt dies üblicherweise eine Syntax), dann kann man oft eine Baumstruktur einsetzen. Ein einfaches Beispiel einer Baumstruktur ist der Stammbaum, wie er in Bild 9-7 an einem einfachen Beispiel verdeutlicht ist. Man sieht, daß Schmidt zwei Kinder hatte: einen Sohn Robert und eine Tochter Anna. Anna hatte drei Kinder: Max, Fritz und Inge. Max seinerseits hatte zwei Kinder: Hans und Paul. Robert, in der linken Hälfte der Darstellung, hatte dagegen keine Nachfahren.

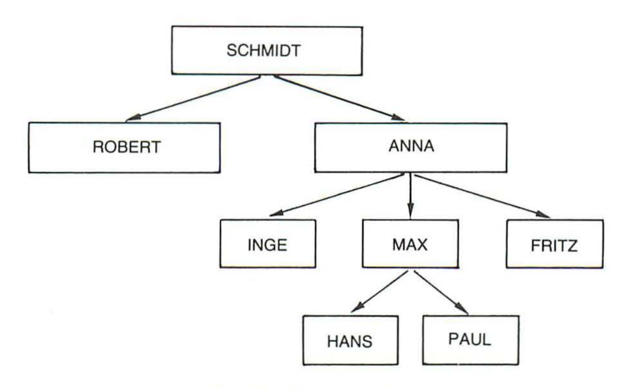


Bild 9-7: Ein Stammbaum

Dies ist eine Baumstruktur. Wir sind im übrigen bereits bei Bild 9-2 auf einen einfachen Baum gestoßen. Die Verzeichnisstruktur dort bildete einen Baum mit zwei Ebenen. Bäume haben ihre Vorzüge überall da, wo Elemente in einer festen Struktur klassifiziert werden müssen. Das ermöglicht Einfügen und Wiedergewinnen von Strukturelementen. Außerdem können Bäume Informationsgruppen gliedern. Eine derartige Aufarbeitung der Information kann für die spätere Verarbeitung – etwa beim Entwurf von Compilern oder Interpretern – notwendig sein.

Doppelt verkettete Listen

Man kann in einer verketteten Liste weitere Verkettungen einführen. Das einfachste Beispiel sind doppelt verkettete Listen. Bild 9-8 bringt ein Beispiel dazu. Wir sehen dort die übliche Verbindungskette von links nach rechts laufen, dazu aber noch eine zweite Kette in Gegenrichtung. Man erreicht dadurch einen einfacheren Rückbezug von einer gegebenen Stelle aus: Vor- und Rückwärtsschritte sind gleich einfach. Allerdings braucht man dann in jedem Block einen weiteren Zeiger.

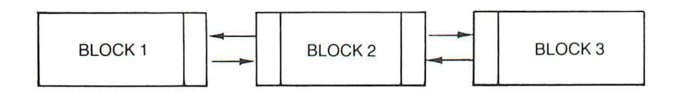


Bild 9-8: Eine doppelt verkettete Liste

Suchen und Sortieren

Suchen oder Sortieren der Elemente einer Liste hängt stark von der benutzten Struktur ab. Für die meisten Datenstrukturen sind viele Suchalgorithmen entwickelt worden. Wir haben bereits die indizierte Adressierung besprochen. Sie ist sinnvoll, wenn die Tabellenelemente bereits nach bekannten Gesichtspunkten geordnet sind. Man kann auf derartige Elemente dann nach ihrer Platznummer zugreifen.
Sequentielles Suchen liegt vor, wenn man sich linear durch einen Block auf der Suche nach einem Element hindurch arbeiten muß. Das ist ganz offensichtlich keine leistungsfähige Methode, doch immer noch besser als gar nichts, wenn die Blockelemente keiner sichtbaren Ordnung unterliegen.
Binäres oder logarithmisches Suchen setzt sich zum Ziel, ein Element in einer sortierten Liste so rasch wie möglich aufzufinden, indem bei jedem Suchschritt das Intervall halbiert wird. Nehmen wir als Beispiel an, wir hätten eine alphabetisch geordnete Liste zu durchsuchen. Man kann dann in der Mitte der Tabelle nachsehen, ob der gedruckte Name in der oberen oder in der unteren Hälfte stehen muß. Ist er in der oberen Hälfte, dann gehen wir dort wieder in die Mitte und sehen nach, ob der Name hier am oberen oder unteren Teil liegt. Und so geht das weiter, bis der gewünschte Wert gefunden ist. Die Suche in einer N Elemente umfassenden Tabelle dauert dann garantiert nicht länger als $\log_2 N$.
Es gibt noch viele andere Suchalgorithmen, auf die wir hier nicht eingehen können.

Zusammenfassung

Dieser Abschnitt sollte lediglich eine kurze Darstellung der verschiedenen in einem Programm verwendbaren Datenstrukturen geben. Obwohl die meisten Datenstrukturen klassifiziert und mit Namen versehen sind, können die Daten in einem komplexen System jede nur denkbare Kombination solcher Strukturen für den gegebenen Anwendungsfall erfinden. Die Möglichkeiten sind hier nur durch den Einfallsreichtum des Programmierers begrenzt. Ganz entsprechend ist eine Vielzahl Such- und Sortiertechniken gut erforscht worden, die den gebräuchlichen Datenstrukturen angepaßt sind. Eine tiefgehendere Beschreibung liegt jedoch außerhalb des Rahmens unseres Buchs hier. Dieses Kapitel hier sollte in erster Linie die Notwendigkeit deutlich machen, daß für die handzuhabenden Daten passende Strukturen erstellt werden müssen, und es sollten die hierfür nötigen Grundwerkzeuge bereitgestellt werden.

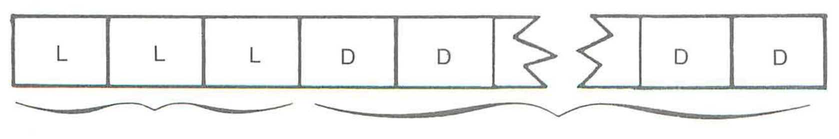
KAPITEL 9
DATENSTRUKTUREN
Teil 2: BEISPIELE

Einleitung

Wir werden hier für die häufigsten Datenstrukturen Tabelle, verkettete Liste und geordneter Baum praktisch verwendbare Beispiele betrachten. Für diese Strukturen sollen Sortier-, Such- und Einfügealgorithmen entwickelt werden. Außerdem werden weiterführende Techniken wie Hashing und Merging beschrieben.
Wenn Sie an diesen weiterführenden Techniken interessiert sind, sollten Sie die dort aufgeführten Beispiele sorgfältig studieren. Dem Anfänger sei jedoch geraten, die Abschnitte über Hashing und Merging zunächst zu überspringen und auf sie zurückzukommen, wenn ein aktuelles Bedürfnis für weiterführende Techniken vorhanden ist.
Um den hier vorgestellten Beispielen folgen zu können, muß der in Teil 1 dieses Kapitels eingeführte Stoff vollständig verstanden sein. Die Programme werden des weiteren alle beim 6502 möglichen Adressierungsarten und viele der in den vorigen Kapiteln vorgestellten Konzepte und Techniken einsetzen.
Wir werden jetzt vier Strukturen vorstellen: eine einfache verkettete Liste, eine verkettete alphabetische Liste mit Verzeichnis und einen Baum. Für jede dieser Strukturen werden drei Programme erstellt: Ein Suchprogramm, ein Einfüge- und ein Löschprogramm.
Außerdem werden am Kapitelende drei spezielle Algorithmen eingeführt, nämlich Hashing, Bubble-Sort und Merging.

Datenformat der Listen

Sowohl für die einfachste als auch die alphabetische Liste werden für die Listenelemente folgendes Datenformat verwenden:



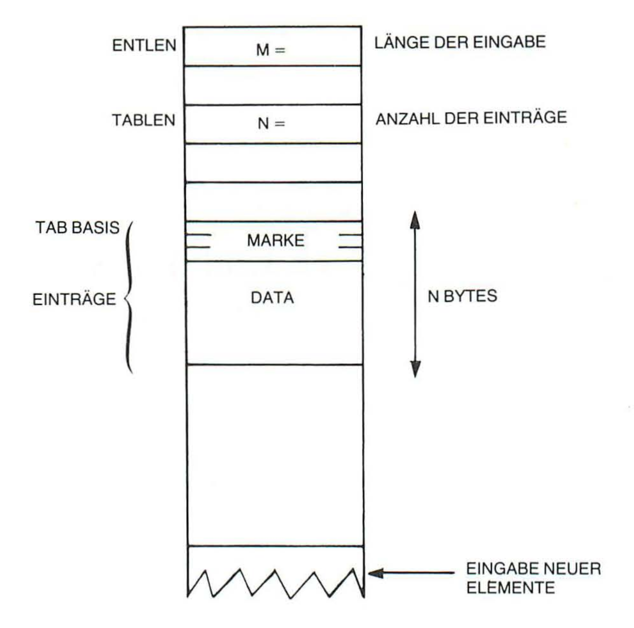


Bild 9-9: Die Tabellenstruktur

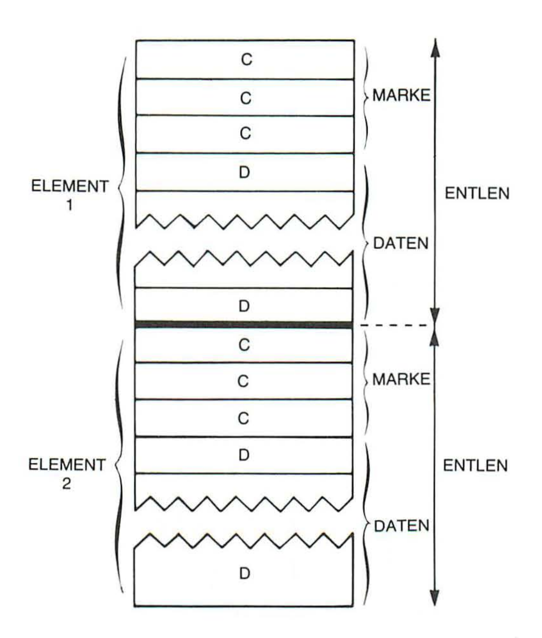


Bild 9-10: Typische Anordnung einer Liste im Speicher

Jedes Listenelement („Eintrag“) besteht aus einem drei Bytes langen Label und einem 1 bis 253 Bytes langen Datenblock. So belegt jeder Eintrag höchstenfalls eine Speicherseite (256 Bytes). Innerhalb einer gegebenen Liste haben alle Elemente dieselbe Länge (vgl. Bild 9-10). Das Programm arbeitet mit diesen Listen mit Hilfe einiger allgemeiner Variablenvereinbarungen:
ENTLEN (element lenght) ist die Länge eines Listenelements. Wenn z.B. jedes Element 10 Datenbytes umfaßt, hat ENTLEN den Inhalt 3 + 10 = 13.
TABASE (Tabellenanfang) gibt den Anfang (die Basis) der Liste oder Tabelle im Speicher an.
POINTR bezeichnet das gerade bearbeitete Element.
OBJECT ist der gerade einzufügende oder zu löschende Eintrag.
TABLEN (Tabellenlänge) gibt die Zahl der Einträge an.
Es wird angenommen, daß kein Label doppelt vorkommt. Abweichen von dieser Übereinkunft erfordert einige kleine Änderungen in den Programmen.

Eine einfache Liste

Die einfache Liste ist als Tabelle mit N Elementen organisiert (Bild 9-11). Die Elemente sind nicht sortiert.
Zur Suche nach einem bestimmten Element muß die ganze Tabelle durchgegangen werden, bis entweder der gesuchte Eintrag gefunden oder das Tabellenende erreicht ist. Das Einfügen geschieht durch Anhängen der neuen Einträge an die bereits vorhandenen. Zum Löschen werden – so vorhanden – die Einträge in den höheren Speicherstellen nach unten geschoben, um die Tabelle kompakt zu halten.

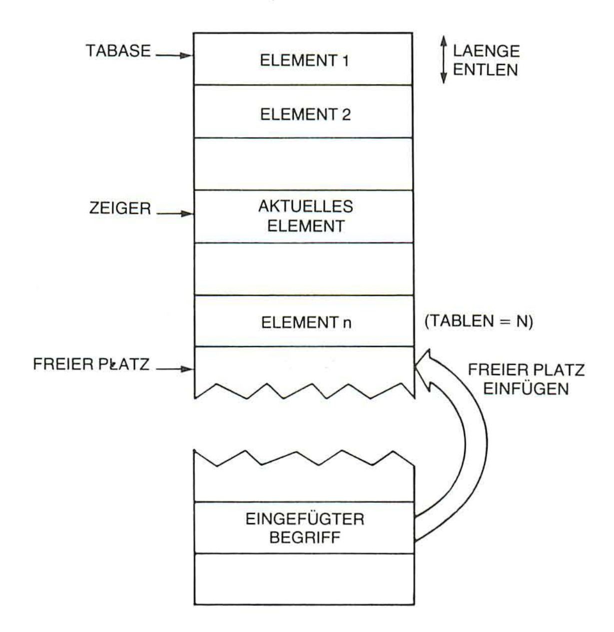


Bild 9-11: Eine einfache Liste

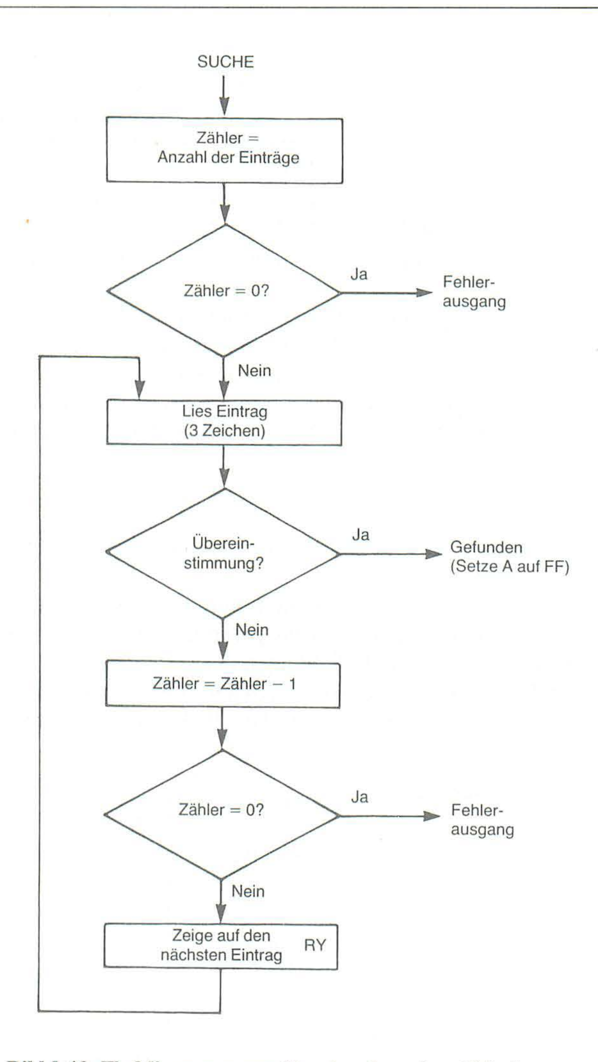


Bild 9-12: Flußdiagramm zum Durchsuchen einer Tabelle

Suchen

Wir setzen eine serielle Suchtechnik ein. Das Labelfeld jedes Eintrags wird mit dem des aufzusuchenden OBJECTs Zeichen für Zeichen verglichen.
Der Arbeitszeiger POINTR wird auf den Tabellenanfang TABASE initialisiert.
Indexregister X erhält die in TABLEN festgehaltene Zahl der Tabelleneinträge als Anfangswert. Die Suche vollzieht sich ohne besondere Tricks. Das Flußdiagramm steht in Bild 9-12. Das dazugehörige Programm in Bild 9-16 (Programm „SEARCH“).

Einfügen eines Elements

Zum Einfügen eines neuen Tabellenelements wird der erste am Listenende verfügbare Speicherblock der Länge ENTLEN verwendet (vgl. Bild 9-11).
Das Programm testet zuerst, ob der neue Eintrag nicht bereits in der Tabelle vorliegt. (Kein Label soll in diesem Beispiel zweimal vorkommen.) Falls nicht, wird die Tabellenlänge TABLEN weitergezählt und OBJECT an das Ende der Liste geschoben. Das Flußdiagramm dazu zeigt Bild 9-13. Das zugehörige Programm findet sich in Bild 9-16 unter dem Namen „NEW". Es steht in Speicherstelle 0636 bis 0659.

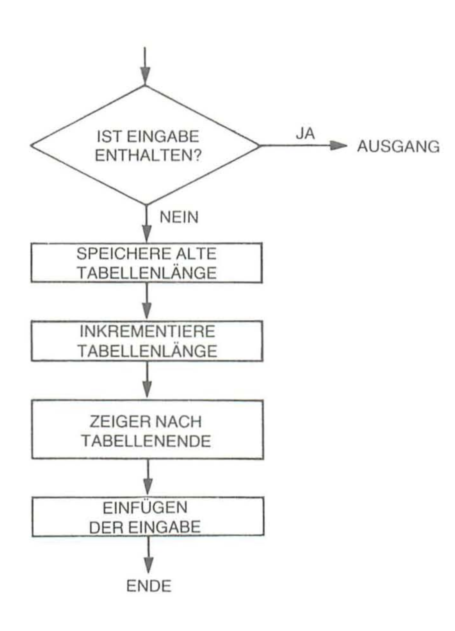


Bild 9-13: Flußdiagramm zum Einfügen in eine Tabelle

Löschen eines Eintrags

Um einen Eintrag aus der Liste zu entfernen, werden die nach ihm im Speicher stehenden Elemente um einen Platz nach oben geschoben und die Angabe zur Tabellenlänge dekrementiert, wie es Bild 9-14 deutlich macht.

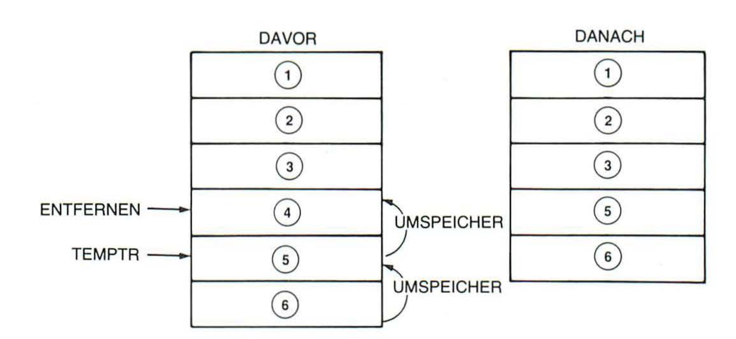


Bild 9-14: Löschen eines Elements aus einer einfachen Liste

Das zugehörige Programm dürfte keine Schwierigkeiten bereiten. Es hat in Bild 9-16 den Namen „DELETE" und steht im Speicher von 0659 bis 0686. Das Flußdiagramm dazu ist in Bild 9-15 wiedergegeben.
Speicherstelle TEMPTR (temporary pointer) wird als Hilfszeiger bei der Verschiebung benutzt. Sie zeigt auf das aufwärts zu schiebende Tabellenelement.
Indexregister Y wird auf die Länge eines Tabellenelements gesetzt und dient zur Automatisierung der Blockverschiebungen. Beachten Sie, daß dazu indirekt-indizierte Adressierung benutzt wird.

```
(0672)      SCHLEIFE    DEY
                        LDA  (TEMPTR), Y
                        STA  (POINTR), Y
                        CPY  #0
                        BNE  SCHLEIFE
```

Während der Übertragung zeigt POINTR immer auf das „Loch" in der Liste, d.h. gibt das Ziel des zu verschiebenden Blocks an.
Die Z-Flagge dient beim Rücksprung als Anzeige einer erforderlichen Löschoperation.

Alphabetische Liste

In der alphabetischen Liste, einer „Tabelle" wie die vorhergehende, liegen die Einträge alle alphabetisch geordnet vor. Das erlaubt den Einsatz schnellerer Suchmethoden als im vorigen Beispiel. Wir werden hier eine binäre Suche durchführen.

Suchen

Der verwendete Suchalgorithmus ist eine klassische binäre Suche. Erinnern Sie sich, daß diese Technik in den Grundzügen der Suche nach einem Namen im Telefonbuch entspricht. Man beginnt irgendwo in der Mitte und geht dann je nach gefundenem Eintrag nach vorne oder nach hinten im Buch weiter. Die Methode ist schnell und genügend einfach als Programm zu verwirklichen.

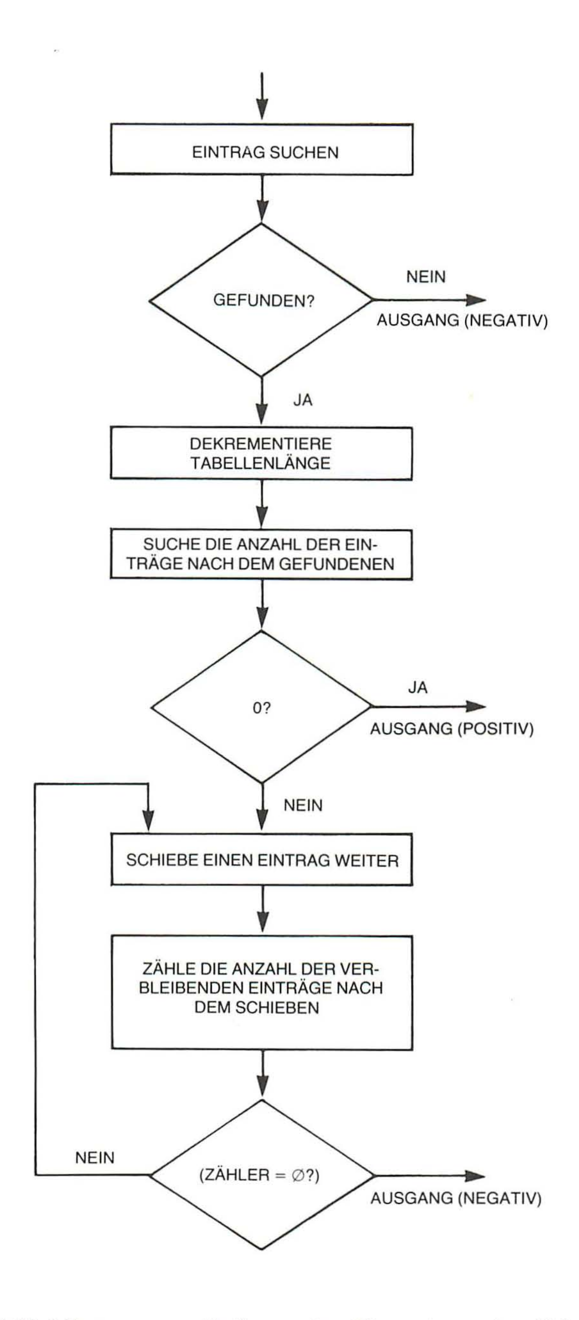


Bild 9-15: Flußdiagramm zum Entfernen eines Elements aus einer Tabelle

```
LOC  CODE          LINE

                   ; SUCHEN, EINFUEGEN UND LOESCHEN IN EINER EINFACHEN LISTE
                   ;
0000               TABASE   = $10          ; LISTENANFANG
0000               POINTR   = $12          ; LAUFZEIGER
0000               TABLEN   = $14          ; LISTENLAENGE
0000               OBJECT   = $15          ; ZEIGER AUF ZU SUCHENDES/LOESCHENDES/EINZUFUEGENDES
0000                                       ;  LISTENELEMENT
0000               ENTLEN   = $17          ; LAENGE EINES LISTENEINTRAGS
0000               TEMPTR   = $18          ; HILFSZEIGER
                   ;
0000                        * = $600
                   ;
0600  A5 10        SEARCH   LDA TABASE     ; INITIALISIERE POINTR AUF LISTENANFANG
0602  85 12                 STA POINTR
0604  A5 11                 LDA TABASE+1
0606  85 13                 STA POINTR+1
0608  A6 14                 LDX TABLEN     ; TABLEN = 0 ?
060A  F0 29                 BEQ OUT        ; JA, FERTIG !
060C  A0 00        ENTRY    LDY #0         ; PRUEFE DIE JEWEILS ERSTEN BUCHSTABEN AUF GLEICHHEIT.
060E  B1 15                 LDA (OBJECT),Y
0610  D1 12                 CMP (POINTR),Y
0612  D0 0E                 BNE NOGOOD     ; UNGLEICH !
0614  C8                    INY            ; GLEICH! PRUEFE ZWEITE BUCHSTABEN.
0615  B1 15                 LDA (OBJECT),Y
0617  D1 12                 CMP (POINTR),Y
0619  D0 07                 BNE NOGOOD     ; UNGLEICH
061B  C8                    INY            ; GLEICH! PRUEFE DRITTE BUCHSTABEN.
061C  B1 15                 LDA (OBJECT),Y
061E  D1 12                 CMP (POINTR),Y
0620  F0 11                 BEQ FOUND      ; EINTRAG GEFUNDEN, FALLS GLEICH; FERTIG !
0622  CA           NOGOOD   DEX            ; BUCHSTABEN VERSCHIEDEN, WEITERE EINTRAEGE VORHANDEN ?
0623  F0 10                 BEQ OUT        ; NEIN, SUCHE ERFOLGLOS.
0625  A5 17                 LDA ENTLEN     ; JA, SETZE ZEIGER AUF NAECHSTEN LISTENEINTRAG.
0627  18                    CLC
0628  65 12                 ADC POINTR
062A  85 12                 STA POINTR
062C  90 DE                 BCC ENTRY
062E  E6 13                 INC POINTR+1
0630  4C 0C 06              JMP ENTRY
0633  A9 FF        FOUND    LDA #$FF       ; GESUCHTER EINTRAG GEFUNDEN, SETZE Z FLAG ZURUECK.
0635  60           OUT      RTS
                   ;
                   ;
                   ;
0636  20 00 06     NEW      JSR SEARCH     ; OBJECT BEREITS IN LISTE VORHANDEN ?
0639  D0 1D                 BNE OUTE       ; JA, FERTIG !
063B  A6 14                 LDX TABLEN     ; LISTE LEER ?
063D  F0 0B                 BEQ INSERT     ; JA.
063F  A5 12                 LDA POINTR     ; NEIN, ZEIGER AUF LISTENENDE SETZEN.
0641  18                    CLC
0642  65 17                 ADC ENTLEN
0644  85 12                 STA POINTR
0646  90 02                 BCC INSERT
0648  E6 13                 INC POINTR+1
064A  E6 14        INSERT   INC TABLEN     ; LISTENLAENGE HERAUFZAEHLEN UND ...
064C  A0 00                 LDY #0         ;  OBJECT AN LISTENENDE ...
064E  A6 17                 LDX ENTLEN     ;   ANFUEGEN.
0650  B1 15        LOOP     LDA (OBJECT),Y ; OBJECT ABSPEICHERN ...
0652  91 12                 STA (POINTR),Y
0654  C8                    INY
```

Bild 9-16: Programme für eine einfache Liste: Suchen, Einfügen, Löschen

```
LINE #  LOC    CODE          LINE

        0655   CA                      DEX
        0656   D0  F8                  BNE LOOP
        0658   60            OUTE      RTS             ; NACH ERFOLGREICHER ABARBEITUNG IST Z FLAG GESETZT.
                             ;
                             ;
                             ;
        0659   20  00  06    DELETE    JSR SEARCH      ; OBJECT BEREITS IN LISTE VORHANDEN ?
        065C   F0  2D                  BEQ OUTS        ; NEIN, FERTIG !
        065E   C6  14                  DEC TABLEN      ; JA, LISTENLAENGE HERUNTERZAEHLEN.
        0660   CA                      DEX             ; LIEGT EINTRAG AM LISTENENDE ?
        0661   F0  26                  BEQ DONE        ; JA, FERTIG.
        0663   A5  12        ADDEM     LDA POINTR      ; NEIN, SETZE HILFSZEIGER AUF NAECHSTEN LISTENEINTRAG.
        0665   18                      CLC
        0666   65  17                  ADC ENTLEN
        0668   85  18                  STA TEMPTR
        066A   A9  00                  LDA #0
        066C   65  13                  ADC POINTR+1     ; CARRY AUF HOEHERWERTIGES BYTE ADDIEREN.
        066E   85  19                  STA TEMPTR+1
        0670   A4  17                  LDY ENTLEN
        0672   88            LOOPE     DEY
        0673   B1  18                  LDA (TEMPTR),Y  ; EIN BYTE UMSPEICHERN.
        0675   91  12                  STA (POINTR),Y
        0677   C0  00                  CPY #0
        0679   D0  F7                  BNE LOOPE       ; EINTRAG NOCH NICHT KOMPLETT VERSCHOBEN, WEITER.
        067B   CA                      DEX             ; ALLE FOLGEEINTRAEGE VERSCHOBEN ?
        067C   F0  0B                  BEQ DONE        ; JA, FERTIG !
        067E   A5  18                  LDA TEMPTR      ; NEIN, ...
        0680   85  12                  STA POINTR      ;  ZEIGER UM EINEN EINTRAG WEITERSETZEN.
        0682   A5  19                  LDA TEMPTR+1
        0684   85  13                  STA POINTR+1
        0686   4C  63  06              JMP ADDEM
        0689   A9  FF        DONE      LDA #$FF        ; FERTIG! NACH KORREKTER BEARBEITUNG IST Z FLAG GESETZT
        068B   60            OUTS      RTS
                             ;
                             ;
        068C                           .END
```

Bild 9-16: Programme für eine einfache Liste: Suchen, Einfügen, Löschen (Fortsetzung)

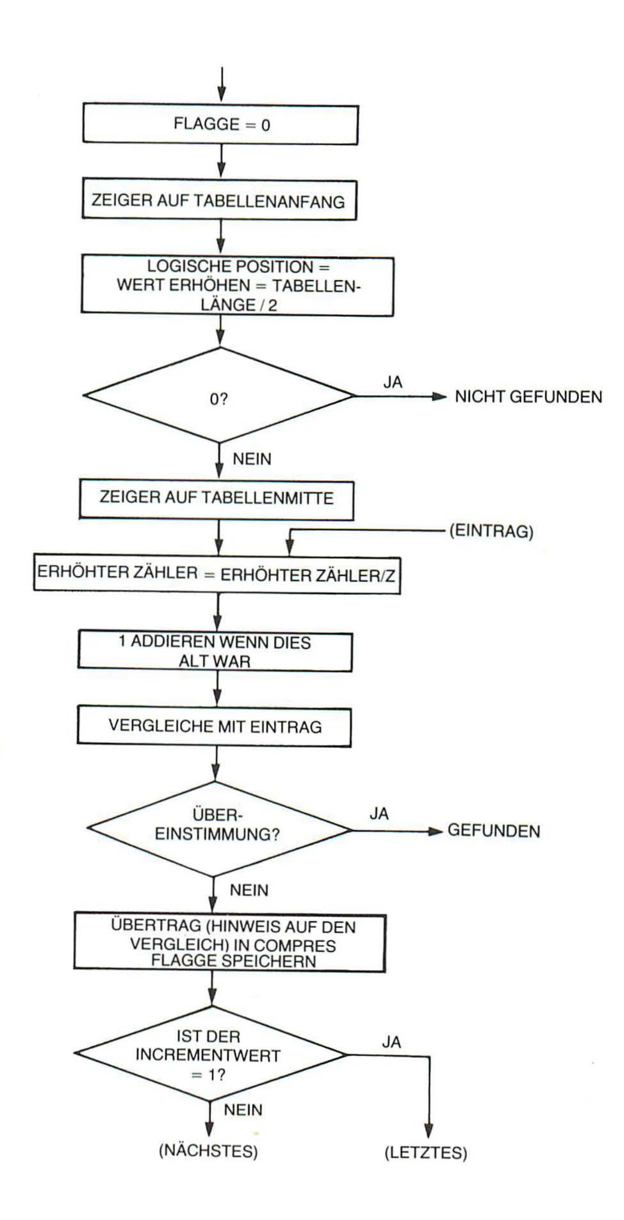


Bild 9-17: Flußdiagramm zur binären Suche

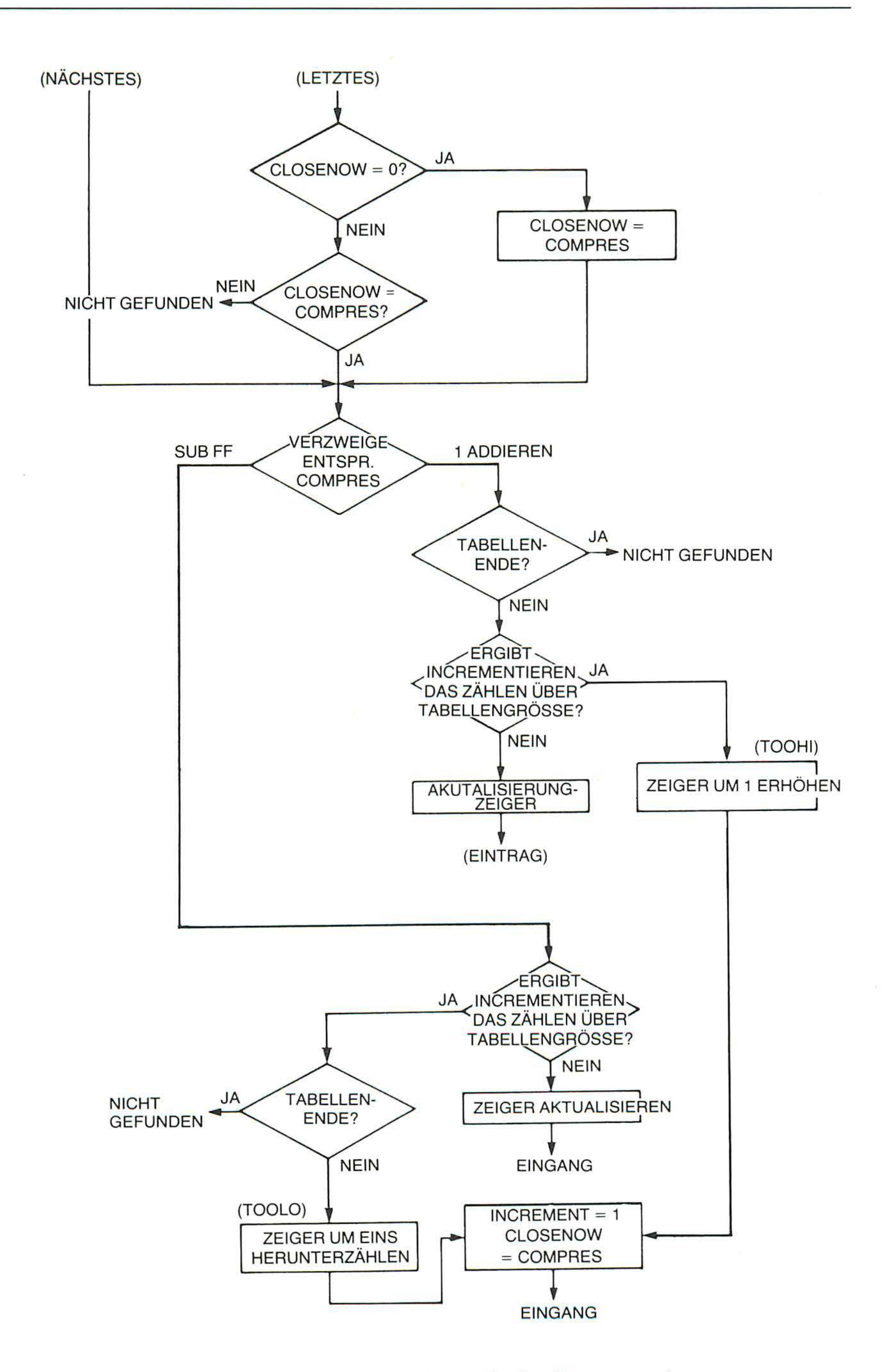


Bild 9-17: Flußdiagramm zur binären Suche (Fortsetzung)

Das Flußdiagramm zur binären Suche zeigt Bild 9-17, das Programm dazu steht in Bild 9-22.
Die Liste enthält die Einträge in alphabetisch geordneter Folge und erreicht sie unter Verwendung binärer („logarithmischer") Suchalgorithmen. Bild 9-18 zeigt dazu ein Beispiel.

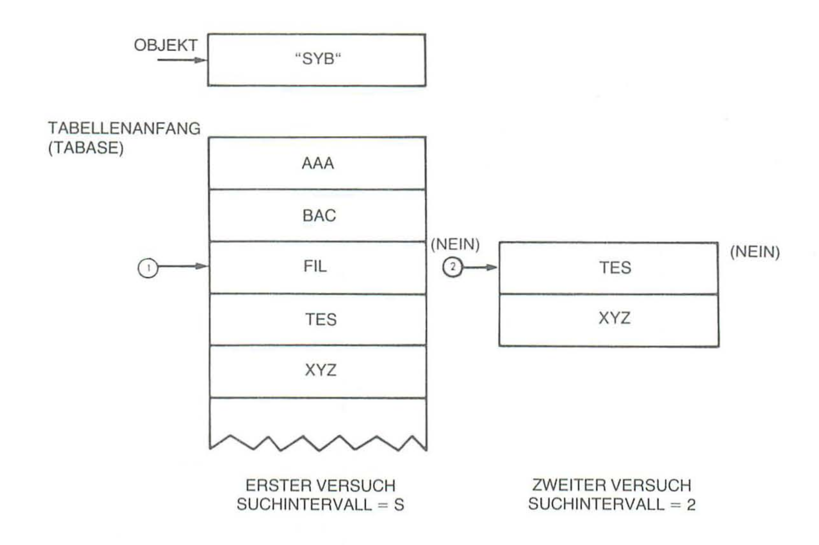


Bild 9-18: Ein binärer Suchvorgang

Die Suche wird etwas kompliziert durch die Notwendigkeit, sich verschiedene Bedingungen merken zu müssen. Das Hauptproblem ist, die Suche nach einem nicht existierenden Objekt zu vermeiden. Andernfalls würden die nebeneinanderliegenden Einträge mit etwas niedrigerem und etwas höherem Wert als der zu suchende auf ewig abwechselnd abgefragt. Um das zu vermeiden, hält eine Flagge im Programm den Wert der Übertragsflagge nach einem negativ verlaufenen Vergleich fest. Wenn der Wert von INCMNT (increment), der angibt, um wieviel der Zeiger im nächsten Schritt weiterzusetzen ist, den Wert „1" erreicht, wird eine andere Flagge namens „CLOSE" (abschließen) auf den Wert dieser Flagge CMPRES (compare result) gesetzt. Da alle weiteren Schritte die Länge „1" haben, hat CMPRES dann nicht mehr denselben Wert wie CLOSE, wenn der Zeiger hinter dem Wort steht, das eigentlich gefunden werden sollte. Diese Eigenschaft gestattet im übrigen der „NEW"-Routine die Feststellung, wo die logischen und physischen Zeiger relativ zum Ort des einzufügenden neuen Objekts stehen.
Wenn das gesuchte OBJECT nicht in der Tabelle steht und der Arbeitszeiger um „1" weitergesetzt wird, wird die CLOSE-Flagge gesetzt. Beim nächsten Durchlauf der Routine ist dann das Vergleichsergebnis gerade umgekehrt. Die beiden Flaggen passen nicht mehr aufeinander, und das Programm kehrt mit der Meldung „nicht gefunden" zurück.

Ein anderes Problem ist die Möglichkeit, daß man beim Addieren oder Subtrahieren des INCMNT-Werts zum Zeiger aus der Tabelle hinausgeraten könnte. Dies wird durch einen „Additions-“ bzw. „Subtraktions“-Test mit dem logischen Zeiger und der Längenangabe der Tabelle gelöst, anstatt die physischen Zeiger zur Entscheidung über die Lage in der Tabelle heranzuziehen.
Wiederholen wir: Zwei Flaggen, CMPRES und CLOSE, dienen dem Programm zum Festhalten bestimmter Informationen. Die CMPRES-Flagge hält fest, ob beim letzten Vergleich die Übertragsflagge C auf „1“ oder auf „0“ gesetzt worden war. Das gibt an, ob das getestete Element kleiner oder größer als das Vergleichselement war. Es war kleiner als dieses, wenn C = 1 ist: CMPRES erhält den Wert „1“. Ist C dagegen gleich „0“, so ist der getestete Eintrag größer als das Vergleichsobjekt, und CMPRES erhält den Wert „FF“.
Beachten Sie weiter, daß im Falle eines negativen Suchergebnisses (C = 1 beim Rücksprung), der Arbeitszeiger auf den Eintrag mit niedrigerem Wert als OBJECT zeigt.
Die zweite vom Programm benutzte Flagge ist CLOSE, diese Flagge erhält den Wert von CMPRES, wenn die Schrittweite INCMNT auf „1“ gesetzt worden ist. Sie dient zum Erkennen, ob das gesuchte Element überhaupt in der Tabelle enthalten ist. Wenn nicht, dann ist CMPRES nach dem nächsten Durchgang nicht mehr gleich CLOSE.
Die anderen vom Programm verwendeten Variablen lauten:

LOGPOS (logical position), die den logischen Ort (Elementnummer) in der Tabelle angibt.
INCMNT (increment) gibt den Wert an, um den der Arbeitszeiger weiter- oder zurückgesetzt wird, wenn der nächste Vergleich negativ ausfällt.
TABLEN (table length) enthält wie üblich die Gesamtlänge der Tabelle (Zahl der Einträge). LOGPOS und INCMNT werden mit TABLEN zum Test auf Einhaltung der Tabellengrenzen verglichen.
Das „SEARCH“ genannte Programm nimmt in Bild 9-22 die Speicherstellen 0600 bis 06E3 ein und sollte sorgfältig studiert werden, da es sehr viel komplexer als das für die lineare Suche geschriebene ist.
Eine zusätzliche Komplikation ergibt sich aus der Tatsache, daß das Suchintervall sowohl gerade als auch ungerade sein kann. Ist es gerade, so muß eine Korrektur vorgenommen werden. Man kann nicht in die Mitte einer Liste aus (beispielsweise) vier Elementen zeigen.
Ist das Intervall ungerade, so wird das Mittenelement durch einen „Trick“ ermittelt: Die Division durch 2 erfolgt durch eine Rechtsverschiebung. Das nach dem LSR-Befehl in den Übertrag C geschobene Bit ist eine „1“, wenn das Intervall ungerade war. Es wird einfach zum Zeiger zurückaddiert:

```
(0615)      DIV  LSR  A         DURCH ZWEI TEILEN
                 ADC  # 0       C-BIT ADDIEREN
                 STA  LOGPOS    NEUER ZEIGER
```

Das OBJECT wird dann mit dem Eintrag in der Mitte des Suchintervalls verglichen. Sind beide gleich, so kehrt das Programm zurück. Andernfalls („NOGOOD“) wird C auf „0“ gesetzt, wenn OBJECT kleiner als der Eintrag ist. Hat INCMNT den Wert

„1“, so wird die (vorher auf „0“ initialisierte) CLOSE-Flagge getestet, ob sie gesetzt worden ist. Wenn nicht, so geschieht das jetzt. Ist sie aber bereits gesetzt, so wird nachgeprüft, ob die Stelle, in der OBJECT eigentlich stehen sollte, bereits überschritten worden ist (in diesem Fall ist es nicht in der Liste vorhanden).

Einfügen eines Elements

Um ein neues Element in die Tabelle einzufügen, wird diese erst einmal (binär) durchsucht, ob es bereits vorhanden ist. In diesem Fall braucht nichts eingefügt zu werden. (Wir nehmen hier an, daß kein Element doppelt vorkommen kann.) Ist das Element in der Tabelle nicht vorhanden, so muß es eingefügt werden. Der Wert der CMPRES-Flagge gibt an, ob es unmittelbar vor oder unmittelbar nach dem letzten getesteten Tabellenelement aufzunehmen ist. Dazu werden alle dem neuen Einfügungsort folgenden Elemente einen Eintrag nach oben geschoben und das neue Element in den frei gewordenen Raum übertragen.
Der Einfügevorgang ist in Bild 9-19 dargestellt, und das zugehörige Programm findet sich unter der Bezeichnung „NEW“ in Bild 9-22, wo es die Adressen 06E3 bis 075E belegt.

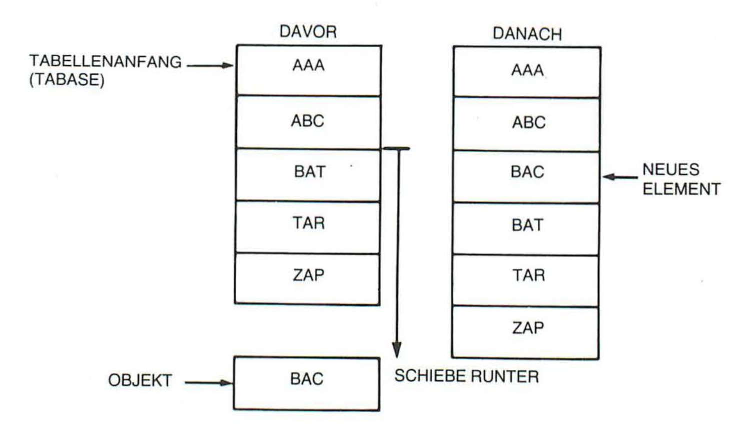


Bild 9-19: Einfügen von „BAT“

Beachten Sie, daß für die Blockverschiebung auch hier wieder indirekt-indizierte Adressierung verwendet worden ist:

```
(073A)              LDY  ENTLEN
          ANDERES   DEY
                    LDA  (POINTR), Y
                    STA  (TEMP), Y
                    CPY  # 0
                    BNE  ANDERES
```

Das gleiche finden Sie unter Speicherstelle 0750.

Löschen eines Elements

Ganz entsprechend nimmt man auch zum Entfernen eines Eintrags zuerst eine binäre Suchoperation vor. Fällt die Suche negativ aus, so braucht nichts gelöscht zu werden. Andernfalls entfernt man das Element, indem man die folgenden Einträge nach oben verschiebt. Ein Beispiel zeigt Bild 9-20. Das Flußdiagramm steht in Bild 9-21 und das Programm namens „DELETE“ (LÖSCHEN) ist in Bild 9-21 zu finden, wo es den Speicherbereich von 075F bis 0799 einnimmt.

Die verkettete Liste

Wir nehmen an, daß die verkettete Liste wie üblich pro Eintrag drei alphanumerische Zeichen für das Label trägt, gefolgt von 1 bis 250 Datenbytes, denen ein 2 Bytes langer Zeiger folgt. Den Abschluß macht eine 1 Byte lange „Belegt“-Markierung. Ist diese auf „1“ gesetzt, so handelt es sich um einen gültigen Eintrag, der von der Einfügeroutine nicht überschrieben werden darf.
Des weiteren haben wir ein Verzeichnis, das für jeden Buchstaben des Alphabets einen Zeiger in der Liste enthält. Dadurch wird der Zugriff auf die Information beschleunigt. Die Labels in den Einträgen sollen aus ASCII-Buchstaben bestehen. Alle am Ende der Liste stehenden Zeiger werden auf einen NIL-Wert gesetzt, der hier gleich dem Tabellenanfang sein soll, da dieser hier nirgends sonst als Listenzeiger auftauchen kann.

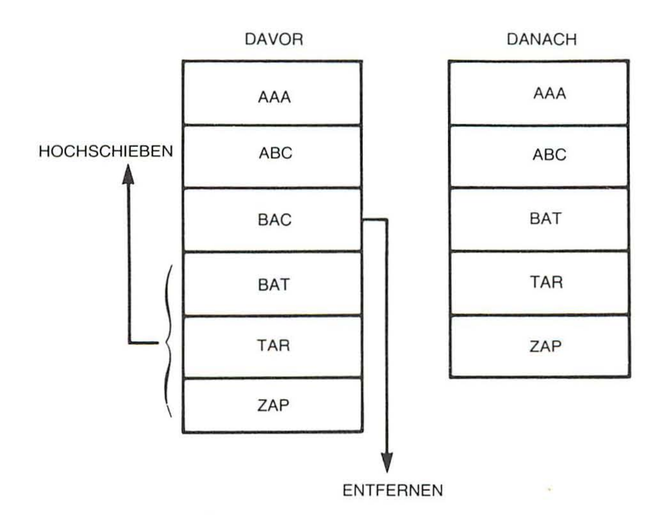


Bild 9-20: Löschen von „BAT“

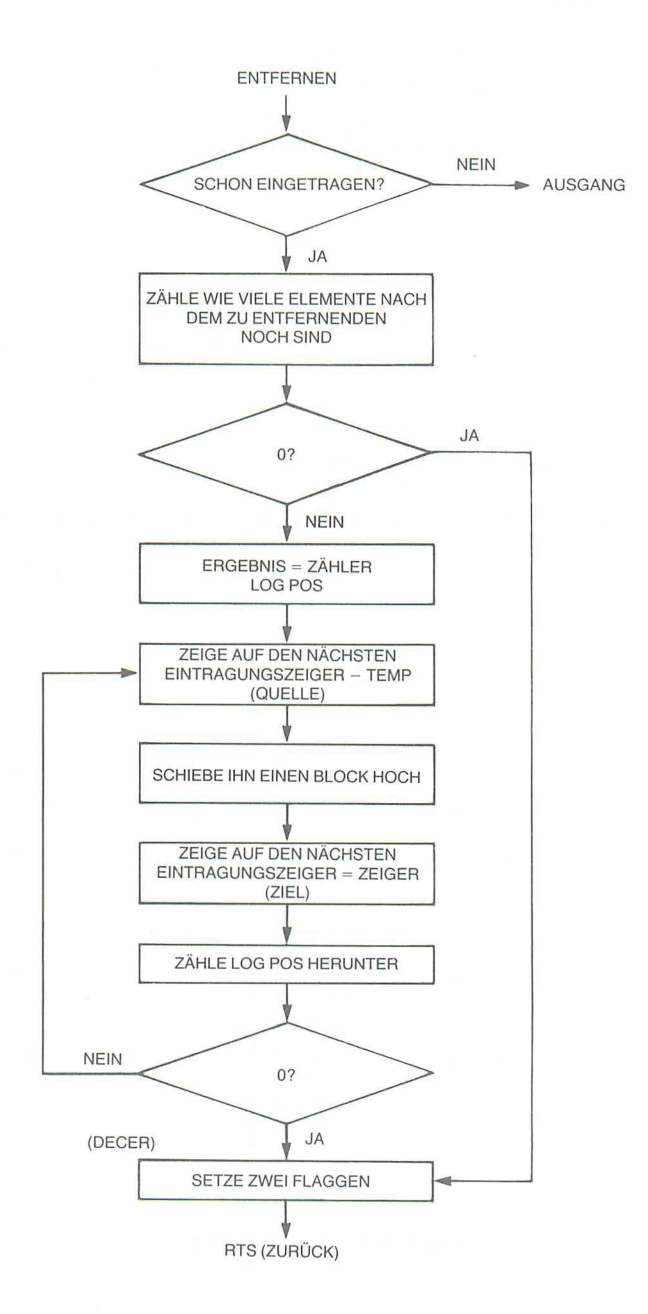


Bild 9-21: Flußdiagramm zum Löschen aus einer alphabetisch sortierten Liste

```
LOC     CODE        LINE

                    ; BINAERE SUCHE, LOESCHEN UND EINFUEGEN
                    ; IN EINER ALPHABETISCH SORTIERTEN LISTE
                    ;
0000                CLOSE   =  $10          ; INTERNES FLAG FUER SEARCH.
0000                CMPRES  =  $11          ; RUECKKEHRSTATUS FUER SEARCH.
0000                TABASE  =  $12          ; LISTENANFANG.
0000                POINTR  =  $14          ; LAUFZEIGER.
0000                TABLEN  =  $16          ; LISTENLAENGE.
0000                LOGPOS  =  $17          ; LOGISCHE POSITION IN DER LISTE.
0000                INCMNT  =  $18          ; INKREMENT FUER BINAERE SUCHE.
0000                TEMP    =  $19          ; HILFSZEIGER.
0000                ENTLEN  =  $1B          ; GROESSE EINES LISTENEINTRAGS.
0000                OBJECT  =  $1C          ; ZEIGER AUF ZU SUCHENDEN/LOESCHENDEN BZW.
0000                                        ;  EINZUFUEGENDEN EINTRAG.
                    ;
0000                        * = $600
                    ;
0600    A9 00       SEARCH  LDA #0          ; FLAGS RUECKSETZEN ...
0602    85 10               STA CLOSE
0604    85 11               STA CMPRES
0606    A5 12               LDA TABASE      ; UND ZEIGER INITIALISIEREN.
0608    85 14               STA POINTR
060A    A5 13               LDA TABASE+1
060C    85 15               STA POINTR+1
060E    A5 16               LDA TABLEN      ; TABELLENLAENGE...
0610    D0 03               BNE DIV         ; UNGLEICH 0, LOGISCHE POSITION SUCHEN.
0612    4C E0 06            JMP OUT         ; GLEICH 0, FERTIG.
0615    4A          DIV     LSR A           ; TABELLENLAENGE DURCH 2 DIVIDIEREN.
0616    69 00               ADC #0          ; HERAUSGESCHOBENES BIT AUFADDIEREN.
0618    85 17               STA LOGPOS      ; LOGISCHE POSITION
061A    85 18               STA INCMNT      ;  UND INKREMENT BELEGEN.
061C    A6 17               LDX LOGPOS      ; ENTLEN MIT LOGPOS MULTIPLIZIEREN
061E    CA                  DEX             ;  UND ZU POINTR ADDIEREN...
061F    F0 0E               BEQ ENTRY
0621    A5 1B       LOOP    LDA ENTLEN
0623    18                  CLC
0624    65 14               ADC POINTR
0626    85 14               STA POINTR
0628    90 02               BCC LOPP
062A    E6 15               INC POINTR+1
062C    CA          LOPP    DEX
062D    D0 F2               BNE LOOP
062F                                        ; POINTR WEIST AUF ANFANG DES GEWUENSCHTEN
062F                                        ;  LISTENABSCHNITTS.
062F    A5 18       ENTRY   LDA INCMNT      ; INKREMENT DURCH 2 DIVIDIEREN...
0631    4A                  LSR A
0632    69 00               ADC #0
0634    85 18               STA INCMNT
0636    A0 00               LDY #0          ; PRUEFE JEWEILS ERSTE BUCHSTABEN...
0638    B1 1C               LDA (OBJECT),Y
063A    D1 14               CMP (POINTR),Y
063C    D0 11               BNE NOGOOD      ; UNGLEICH !
063E    C8                  INY             ; GLEICH, ZWEITE BUCHSTABEN PRUEFEN...
063F    B1 1C               LDA (OBJECT),Y
0641    D1 14               CMP (POINTR),Y
0643    D0 0A               BNE NOGOOD      ; UNGLEICH !
0645    C8                  INY             ; GLEICH, DRITTE BUCHSTABEN PRUEFEN...
0646    B1 1C               LDA (OBJECT),Y
0648    D1 14               CMP (POINTR),Y
064A    D0 03               BNE NOGOOD      ; UNGLEICH !
```

Bild 9-22: Programme zur alphabetisch sortierten Liste: Binäre Suche, Löschen, Einfügen

```
LOC    CODE           LINE

064C   4C E2 06               JMP FOUND      ; GLEICH, EINTRAG GEFUNDEN, Z FLAG GESETZT !
064F   A0 FF          NOGOOD  LDY #$FF       ; OBJECT UND LISTENEINTRAG NICHT IDENTISCH,
0651                                         ;  Z FLAG RUECKSETZEN.
0651   90 02                  BCC TESTS      ; CARRY FLAG RUECKGESETZT, FALLS OBJECT < POINTR
0653   A0 01                  LDY #1
0655   84 11          TESTS   STY CMPRES
0657   A4 18                  LDY INCMNT
0659   88                     DEY            ; INKREMENT = 1 ?
065A   D0 10                  BNE NEXT       ; NEIN, WEITER !
065C   A5 10                  LDA CLOSE      ; JA! WAR CLOSE - FLAG GESETZT ?
065E   F0 08                  BEQ MAKCLO     ; NEIN, DANN SETZE ES JETZT.
0660   38                     SEC
0661   E5 11                  SBC CMPRES     ; OBJECT DORT, WO ES SICH BEFINDEN MUESSTE NICHT
0663                                         ;  VORGEFUNDEN ?
0663   F0 07                  BEQ NEXT       ; NEIN, SUCHE WEITER.
0665   4C E0 06               JMP OUT        ; JA, FERTIG.
0668   A5 11          MAKCLO  LDA CMPRES     ; CLOSE AUF CMPRES SETZEN.
066A   85 10                  STA CLOSE
066C   24 11          NEXT    BIT CMPRES
066E   30 35                  BMI SUBIT
0670   A5 16                  LDA TABLEN     ; FUEHRT EIN WEITERSETZEN VON POINTR UM INCMNT ...
0672   38                     SEC            ;  UEBER DAS LISTENENDE HINAUS ?
0673   E5 17                  SBC LOGPOS
0675   F0 69                  BEQ OUT        ; LISTENENDE BEREITS ERREICHT.
0677   E5 18                  SBC INCMNT     ;
0679   90 1A                  BCC TOOHI      ; JA, SPEZIELLE BEHANDLUNG DIESES FALLES NOETIG.
067B   A6 18                  LDX INCMNT     ; NEIN, SETZE POINTR UM
067D   A5 1B          ADDER   LDA ENTLEN     ;  ENTLEN MAL INCMNT BYTES WEITER...
067F   18                     CLC
0680   65 14                  ADC POINTR
0682   85 14                  STA POINTR
0684   90 02                  BCC AD1
0686   E6 15                  INC POINTR+1
0688   CA             AD1     DEX
0689   D0 F2                  BNE ADDER
068B   A5 17                  LDA LOGPOS     ; LOGISCHE POSITION WEITERSETZEN ...
068D   18                     CLC
068E   65 18                  ADC INCMNT
0690   85 17                  STA LOGPOS
0692   4C 2F 06               JMP ENTRY
0695   E6 17          TOOHI   INC LOGPOS     ; LOGISCHE POSITION HERAUFZAEHLEN.
0697   A5 1B                  LDA ENTLEN     ; POINTR UM EINEN EINTRAG WEITERSETZEN...
0699   18                     CLC
069A   65 14                  ADC POINTR
069C   85 14                  STA POINTR
069E   90 35                  BCC SETCLO
06A0   E6 15                  INC POINTR+1
06A2   4C D5 06               JMP SETCLO
06A5   A5 17          SUBIT   LDA LOGPOS     ; ERREICHT ODER UNTERSCHREITET DAS HERUNTERSETZEN...
06A7   38                     SEC            ;  VON POINTR DEN LISTENANFANG ?
06A8   E5 18                  SBC INCMNT
06AA   F0 17                  BEQ TOOLOW     ; JA,...
06AC   90 15                  BCC TOOLOW     ;  SPEZIELLE BEHANDLUNG ERFORDERLICH !
06AE   85 17                  STA LOGPOS     ; NEIN, NEUE LOGISCHE POSITION FESTHALTEN.
06B0   A6 18                  LDX INCMNT
06B2   A5 14          SUBLOP  LDA POINTR     ; POINTR UM ENTLEN MAL INCMNT BYTES HERUNTERSETZEN ...
06B4   38                     SEC
06B5   E5 1B                  SBC ENTLEN
06B7   85 14                  STA POINTR
06B9   B0 02                  BCS SUBO
```

Bild 9-22: Programme zur alphabetisch sortierten Liste: Binäre Suche, Löschen, Einfügen (Fortsetzung)

```
LOC     CODE            LINE

06BB    C6  15                  DEC POINTR+1
06BD    CA              SUBO    DEX             ; FERTIG ?
06BE    D0  F2                  BNE SUBLOP      ; NEIN, WEITER !
06C0    4C  2F  06              JMP ENTRY       ; JA !
06C3    A6  17          TOOLOW  LDX LOGPOS      ; LOGISCHE POSITION
06C5    CA                      DEX             ;  BEREITS = 1 ?
06C6    F0  18                  BEQ OUT         ; JA, FERTIG.
06C8    C6  17                  DEC LOGPOS      ; NEIN, ZAEHLE SIE HERUNTER.
06CA    A5  14                  LDA POINTR      ; POINTR UM EINEN EINTRAG HERUNTERZAEHLEN...
06CC    38                      SEC
06CD    E5  1B                  SBC ENTLEN
06CF    85  14                  STA POINTR
06D1    B0  02                  BCS SETCLO
06D3    C6  15                  DEC POINTR+1
06D5    A9  01          SETCLO  LDA #1          ; SETZE FLAGS
06D7    85  18                  STA INCMNT
06D9    A5  11                  LDA CMPRES
06DB    85  10                  STA CLOSE
06DD    4C  2F  06              JMP ENTRY
06E0    A2  FF          OUT     LDX #$FF        ; Z FLAG GESETZT, FALLS SUCHE ERFOLGREICH.
06E2    60              FOUND   RTS
                        ;
                        ;
                        ;
06E3    20  00  06      NEW     JSR SEARCH      ; OBJECT BEREITS IN LISTE ENTHALTEN ?
06E6    F0  76                  BEQ OUTE        ; JA, FERTIG !
06E8    A5  16                  LDA TABLEN      ; NEIN; LISTE LEER ?
06EA    F0  62                  BEQ INSERT      ; JA, FUEGE OBJECT EIN.
06EC    24  11                  BIT CMPRES      ; PRUEFE ERGEBNIS DES LETZTEN VERGLEICHS IN SEARCH...
06EE    10  05                  BPL LOSIDE      ; OBJECT MUSS IN UNTEREN LISTENTEIL EINGEFUEGT WERDEN !
06F0    C6  17                  DEC LOGPOS      ; KORRIGIERE LOGPOS FUER SPAETERES SUB...
06F2    4C  00  07              JMP SETUP
06F5    A5  1B          LOSIDE  LDA ENTLEN      ; POINTR HINTER ZIELEINTRAG
06F7    18                      CLC             ;  FUER OBJECT SETZEN...
06F8    65  14                  ADC POINTR
06FA    85  14                  STA POINTR
06FC    90  02                  BCC SETUP
06FE    E6  15                  INC POINTR+1
0700    A5  16          SETUP   LDA TABLEN      ; WIEVIELE EINTRAEGE FOLGEN AUF DEN ZIELEINTRAG ?
0702    38                      SEC
0703    E5  17                  SBC LOGPOS
0705    F0  47                  BEQ INSERT
0707    AA                      TAX
0708    A8                      TAY
0709    88                      DEY             ; STEHT POINTR SCHON AM ENDE DER LISTE ?
070A    F0  0E                  BEQ SETEMP      ; JA.
070C    A5  1B          UPLOOP  LDA ENTLEN      ; NEIN, VERSCHIEBE IHN DORTHIN...
070E    18                      CLC
070F    65  14                  ADC POINTR
0711    85  14                  STA POINTR
0713    90  02                  BCC SETO
0715    E6  15                  INC POINTR+1
0717    88              SETO    DEY
0718    D0  F2                  BNE UPLOOP
071A    A5  14          SETEMP  LDA POINTR      ; ENTLEN ZU POINTR ADDIEREN...
071C    18                      CLC
071D    65  1B                  ADC ENTLEN
071F    85  19                  STA TEMP        ; DIESEN IN TEMP SPEICHERN.
0721    90  01                  BCC SET1
0723    C8                      INY             ; Y REGISTER WAR BEREITS 0 .
```

Bild 9-22: Programme zur alphabetisch sortierten Liste: Binäre Suche, Löschen, Einfügen (Fortsetzung)

```
LOC    CODE          LINE

0724   98            SET1    TYA
0725   18                    CLC
0726   65 15                 ADC POINTR+1
0728   85 1A                 STA TEMP+1
072A   A4 1B         MOVER   LDY ENTLEN        ; ENTLEN MAL
072C   88            ANOTHR  DEY               ;  EIN BYTE
072D   B1 14                 LDA (POINTR),Y    ;   UMSPEICHERN...
072F   91 19                 STA (TEMP),Y
0731   C0 00                 CPY #0
0733   D0 F7                 BNE ANOTHR
0735   A5 14                 LDA POINTR        ; POINTR UND TEMP
0737   38                    SEC               ;  UM ENTLEN
0738   E5 1B                 SBC ENTLEN        ;   HERUNTERZAEHLEN...
073A   85 14                 STA POINTR
073C   B0 02                 BCS M1
073E   C6 15                 DEC POINTR+1
0740   CA            M1      DEX
0741   D0 D7                 BNE SETEMP
0743   A5 1B                 LDA ENTLEN        ; POINTR AUF ZIELEINTRAG FUER OBJECT
0745   18                    CLC               ;  ZURUECKSETZEN...
0746   65 14                 ADC POINTR
0748   85 14                 STA POINTR
074A   90 02                 BCC INSERT
074C   E6 15                 INC POINTR+1
074E   A0 00         INSERT  LDY #0            ; OBJECT IN LISTE EINTRAGEN...
0750   A6 1B                 LDX ENTLEN
0752   B1 1C         INNER   LDA (OBJECT),Y
0754   91 14                 STA (POINTR),Y
0756   C8                    INY
0757   CA                    DEX
0758   D0 F8                 BNE INNER
075A   E6 16                 INC TABLEN        ; LISTENLAENGE HERAUFZAEHLEN.
075C   A2 FF                 LDX #$FF
075E   60            OUTE    RTS               ; BEI ERFOLGREICHEM DURCHLAUF IST Z FLAG RUECKGESETZT.
                     ;
                     ;
                     ;
075F   20 00 06      DELETE  JSR SEARCH        ; SUCHE OBJECT IN LISTE.
0762   D0 35                 BNE OUTS          ; NICHT GEFUNDEN, FERTIG.
0764   A5 16                 LDA TABLEN        ; GEFUNDEN! WIEVIELE EINTRAEGE FOLGEN AUF OBJECT ?
0766   38                    SEC
0767   E5 17                 SBC LOGPOS
0769   F0 2A                 BEQ DECER         ; KEINE, OBJECT STEHT AM LISTENENDE.
076B   85 17                 STA LOGPOS        ; ANZAHL DER FOLGEEINTRAEGE.
076D   A5 1B         BIGLOP  LDA ENTLEN        ; SETZE TEMP EINEN EINTRAG HINTER OBJECT...
076F   18                    CLC
0770   65 14                 ADC POINTR
0772   85 19                 STA TEMP
0774   A9 00                 LDA #0
0776   65 15                 ADC POINTR+1
0778   85 1A                 STA TEMP+1
077A   A6 1B                 LDX ENTLEN        ; ZAEHLER AUF ENTLEN SETZEN.
077C   A0 00                 LDY #0
077E   B1 19         BYTE    LDA (TEMP),Y      ; EIN BYTE UMSPEICHERN...
0780   91 14                 STA (POINTR),Y
0782   C8                    INY
0783   CA                    DEX             . ; EINTRAG VOLLSTAENDIG KOPIERT ?
0784   D0 F8                 BNE BYTE          ; NEIN, WEITER.
0786   A5 1B                 LDA ENTLEN        ; POINTR WEITERSETZEN...
0788   18                    CLC
```

```
LOC    CODE          LINE

0789   65 14                 ADC POINTR
078B   85 14                 STA POINTR
078D   90 02                 BCC D2
078F   E6 15                 INC POINTR+1
0791   C6 17         D2      DEC LOGPOS        ; ALLE EINTRAEGE VERSCHOBEN ?
0793   D0 D8                 BNE BIGLOP        ; NEIN, WEITER.
0795   C6 16         DECER   DEC TABLEN        ; LISTENLAENGE HERUNTERZAEHLEN.
0797   A9 00                 LDA #0            ; Z FLAG NACH ERFOLGREICHER ABARBEITUNG GESETZT.
0799   60            OUTS    RTS
079A                         .END
```

Bild 9-22: Programme zur alphabetisch sortierten Liste: Binäre Suche, Löschen, Einfügen (Fortsetzung)

Einfüge- und Löschprogramme führen die naheliegenden Zeigermanipulationen aus. Sie benutzen die INDEX-Flagge zur Anzeige, ob ein auf ein Objekt weisender Zeiger von einem anderen Listeneintrag oder vom Verzeichnis kommt. Die Programme stehen in Bild 9-27. Bild 9-23 gibt die verwendete Datenstruktur wieder.

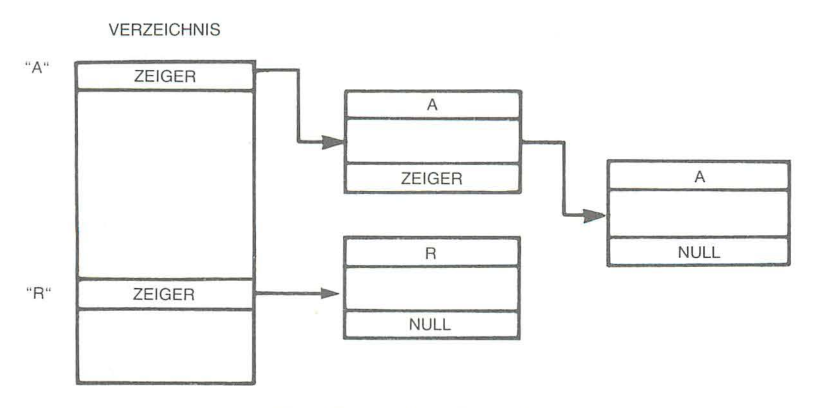
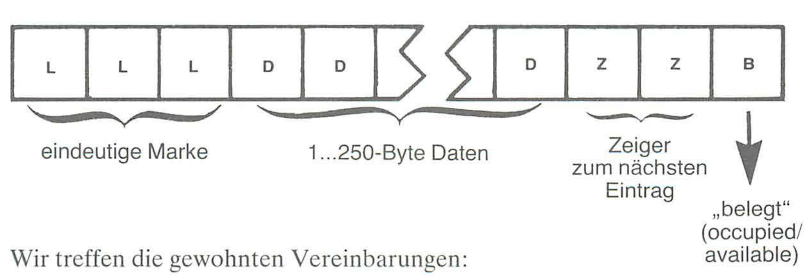


Bild 9-23: Struktur der verketteten Liste

Man kann diese Datenstruktur beispielsweise für ein Adreßverzeichnis verwenden, in dem jede Person durch einen eindeutigen Kode aus drei Buchstaben wiedergegeben ist und das Datenfeld eine vereinfachte Adreßangabe plus Telefonnummer (250 Zeichen insgesamt) enthält.

Sehen wir uns die Struktur aus Bild 9-23 genauer an.

Das Eintragsformat sieht so aus:



Wir treffen die gewohnten Vereinbarungen:

ENTLEN: gesamte Elementlänge (in Bytes)
TABASE: Basisadresse der Liste
TABLEN: Zahl der Einträge (0 bis 256)

Die OBJECT-Adresse soll bei Aufruf des Programms immer in Y stehen.
REFBASE ist ein Zeiger auf die Basisadresse (Anfangsadresse) des Verzeichnisses (der „Referenztabelle“).

Jede Zweibyteadresse im Verzeichnis (directory) zeigt auf den Listeneintrag, in dessen Label der betreffende Buchstabe zum erstenmal auftaucht. Auf diese Art und Weise bilden die Elemente, deren Labels mit demselben Buchstaben beginnen, praktisch eigene Listen in der Gesamtstruktur. Diese Eigenschaft vereinfacht die Suche und entspricht der Situation in einem Adreßbuch. Beachten Sie, daß während der Einfüge- bzw. Löschoperation keinerlei Daten verschoben werden. Es werden ausschließlich Zeiger ausgetauscht, wie es zu einer richtigen verketteten Liste gehört.
Wenn zu einem bestimmten Buchstaben kein Eintrag gefunden wurde oder wenn nach einem Eintrag kein weiterer folgt, so deutet der zugeordnete Zeiger auf den Tabellenanfang (= NIL). Vereinbarungsgemäß soll am Boden der Tabelle ein Wert gespeichert werden, dessen absolute Differenz zu „Z" größer ist als die Differenz zwischen „A" und „Z", und der so eine Tabellenmarkierung darstellt (EOT – end of table). Wir nehmen hier an, daß der EOT-Wert einen ganzen Eintrag einnimmt. Es kann aber genausogut ein einzelner Wert dafür verwendet werden.
Die hier verwendeten Buchstaben sollen ASCII-kodiert sein. Falls dies geändert werden soll, muß die Konstante in der PRETAB-Routine anders gefaßt werden.
Die Tabellenendmarkierung EOT wird auf den Anfang der Tabelle (NIL) gesetzt. Weiter sei vereinbart, daß die „NIL-Zeiger", die entweder am Ende einer Zeichenkette oder im Verzeichnis (wenn kein zugehöriger Eintrag vorliegt) stehen, zur eindeutigen Kennzeichnung auf den Wert der Tabellenbasis gesetzt werden. Man könnte auch andere Vereinbarungen treffen. Insbesondere könnte man durch eine EOT-Sondervereinbarung Platz einsparen, da dann für nichtvorhandene Einträge kein NIL-Wert angegeben werden muß.
Einfügen und Löschen wird wie üblich durchgeführt (vgl. Teil 1 dieses Kapitels), indem einfach die zugehörigen Zeiger verändert werden. Die INDEX-Flagge gibt an, ob ein Zeiger aus dem Verzeichnis oder von einem anderen Listenelement kommt.

Suchen

Das Suchprogramm SEARCH steht im Speicher von 0600 bis 0650. Es benutzt außerdem das Unterprogramm PRETAB an Adresse 06F8.

Die Suche erfolgt ohne große Tricks:
1. Verzeichniseintrag für den ersten Buchstaben im OBJECT-Label übernehmen.
2. Zeiger entnehmen und damit auf den zugehörigen Eintrag zugreifen. Falls der Zeiger den Wert NIL hat, gibt es kein passendes Element in der Liste.
3. Falls nicht NIL, das angegebene Element mit OBJECT vergleichen. Stimmen beide Labels überein, so ist der gesuchte Eintrag gefunden. Wenn nicht, Zeiger auf den nächsten Eintrag aufsuchen.
4. Zurück zu Schritt 2.

Bild 9-24 zeigt ein Suchbeispiel.

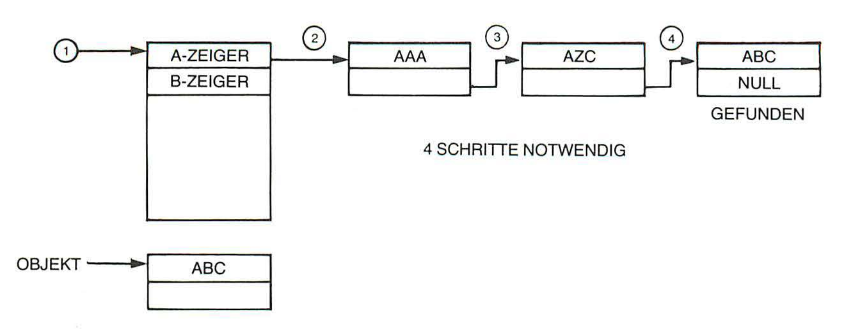


Bild 9-24: Verkettete Liste: Eine Suchoperation

Eintrag einfügen

Die Einfügeoperation ist im Grunde eine Suche nach dem einzufügenden Element. Wenn dabei NIL gefunden wird, muß der Eintrag eingefügt werden. Dazu wird bis zur EOT-Markierung nach einem Speicherblock gesucht, dessen Belegtflagge „Verfügbar" (available) anzeigt. Ansonsten wird nach EOT ein neuer Block bereitgestellt. Das Programm heißt „NEW" und befindet sich in Adresse 0651 bis 06BD. Bild 9-25 zeigt ein Beispiel.

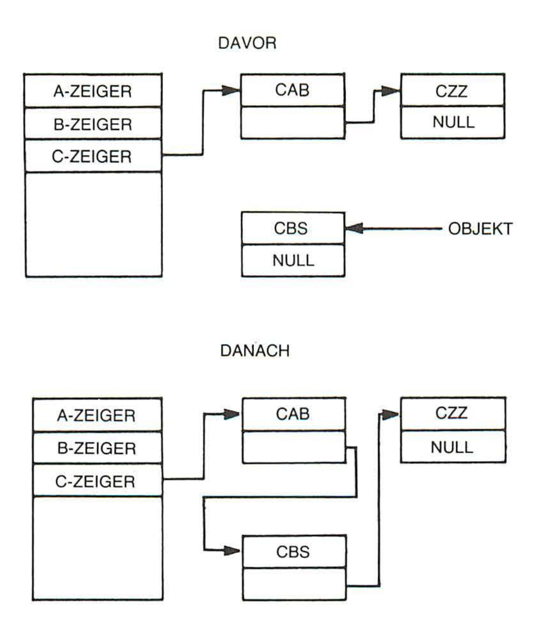


Bild 9-25: Verkettete Liste: Einfügen eines Elements

Eintrag löschen

Ein Eintrag wird gelöscht, indem seine Belegtmarkierung auf „Verfügbar“ (available) gesetzt und der Zeiger im Verzeichnis oder dem vorangehenden Listeneintrag abgepaßt wird. Das Programm heißt „DELETE“ und belegt die Speicherstellen 06BE bis 06F7. Ein Beispiel steht in Bild 9-26.

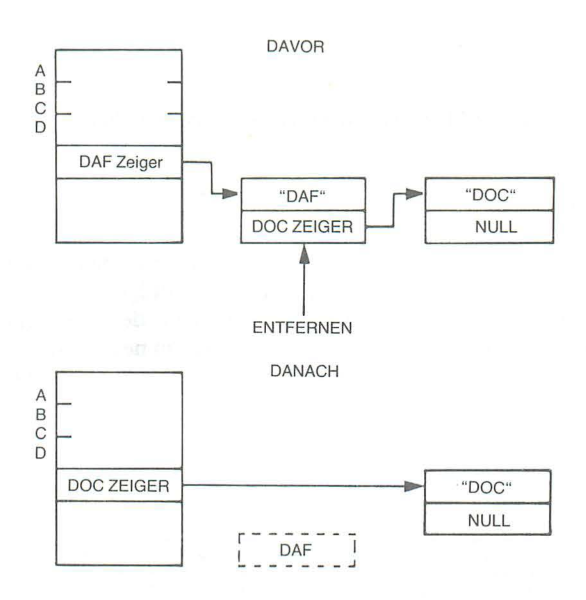


Bild 9-26: Verkettete Liste: Löschen eines Elements

```
LOC     CODE          LINE

                      ; SUCHEN, EINFUEGEN UND LOESCHEN IN EINER VERKETTETEN LISTE
                      ;
0000                  INDEXD   =  $10        ; GLEICH 0, FALLS POINTR VON LISTENELEMENT UEBERNOMMEN.
0000                  INDLOC   =  $11        ; INDEXPOSITION.
0000                  POINTR   =  $13        ; LAUFZEIGER.
0000                  OBJECT   =  $15        ; ZEIGER AUF ZU SUCHENDEN/LOESCHENDEN/EINZUFUEGENDEN
0000                                         ;  EINTRAG.
0000                  TEMP     =  $17        ; HILFSZEIGER.
0000                  REFBAS   =  $19        ; ANFANG DES VERZEICHNISSES.
0000                  OLD      =  $1B        ; ZEIGER AUF DAS DEM VON POINTR REFERIERTEN EINTRAG
0000                                         ;  VORAUSGEHENDE LISTENELEMENT.
0000                  TABASE   =  $1D        ; LISTENANFANG.
0000                  ENTLEN   =  $1F        ; LAENGE EINES EINTRAGS.
                      ;
0000                           * = $600
                      ;
0600    A9 01         SEARCH   LDA #1         ; FLAGS INITIALISIEREN.
0602    85 10                  STA INDEXD
0604    20 F8 06               JSR PRETAB     ; Y-REGISTER ALS ZEIGER AUF ANFANGSEINTRAG SETZEN
0607    B1 11                  LDA (INDLOC),Y ;  UND IN POINTR SPEICHERN...
0609    85 13                  STA POINTR
060B    C8                     INY
060C    B1 11                  LDA (INDLOC),Y
060E    85 14                  STA POINTR+1
0610    A0 00         ENTRY    LDY #0
0612    B1 13                  LDA (POINTR),Y
0614    C9 7C                  CMP #$7C       ; LISTENENDE ERREICHT ?
0616    F0 36                  BEQ NOTFND     ; JA, OBJECT NICHT GEFUNDEN.
0618    B1 15                  LDA (OBJECT),Y ; VERGLEICHE DIE JEWEILS ERSTEN BUCHSTABEN...
061A    D1 13                  CMP (POINTR),Y
061C    90 30                  BCC NOTFND     ; OBJECT > POINTR, OBJECT NICHT GEFUNDEN !
061E    D0 12                  BNE NOGOOD     ; OBJECT < POINTR, WEITER.
0620    C8                     INY
0621    B1 15                  LDA (OBJECT),Y ; VERGLEICHE ZWEITE BUCHSTABEN...
0623    D1 13                  CMP (POINTR),Y
0625    90 27                  BCC NOTFND     ; OBJECT > POINTR, OBJECT NICHT GEFUNDEN !
0627    D0 09                  BNE NOGOOD     ; OBJECT < POINTR, WEITER.
0629    C8                     INY
062A    B1 15                  LDA (OBJECT),Y ; VERGLEICHE DRITTE BUCHSTABEN...
062C    D1 13                  CMP (POINTR),Y
062E    90 1E                  BCC NOTFND     ; OBJECT > POINTR, OBJECT NICHT GEFUNDEN !
0630    F0 1E                  BEQ FOUND      ; OBJECT = POINTR, OBJECT GEFUNDEN, FERTIG !
0632    A5 14         NOGOOD   LDA POINTR+1   ; POINTR FUER EVTL. SPAETEREN GEBRAUCH SPEICHERN...
0634    85 1C                  STA OLD+1
0636    A5 13                  LDA POINTR
0638    85 1B                  STA OLD
063A    A4 1F                  LDY ENTLEN     ; VERKETTUNGSZEIGER AUS EINTRAG
063C    B1 13                  LDA (POINTR),Y ;  IN POINTR KOPIEREN...
063E    AA                     TAX
063F    C8                     INY
0640    B1 13                  LDA (POINTR),Y
0642    85 14                  STA POINTR+1
0644    8A                     TXA
0645    85 13                  STA POINTR
0647    A9 00                  LDA #0
0649    85 10                  STA INDEXD     ; FLAG RUECKSETZEN.
064B    4C 10 06               JMP ENTRY
064E    A9 FF         NOTFND   LDA #$FF
0650    60            FOUND    RTS            ; Z FLAG GESETZT, WENN OBJECT GEFUNDEN WURDE.
                      ;
```

Bild 9-27: Programme zur verketteten Liste

```
LINE #  LOC    CODE           LINE

                          ;
                          ;
0651   20 00 06    NEW    JSR SEARCH      ; OBJECT IN LISTE ENTHALTEN ?
0654   F0 67              BEQ OUTE        ; JA, FERTIG.
0656   A5 1D              LDA TABASE      ; NEIN, SUCHE FREIEN PLATZ IN LISTE...
0658   18                 CLC
0659   69 01              ADC #1          ; EOT UEBERSPRINGEN.
065B   85 17              STA TEMP
065D   A9 00              LDA #0
065F   65 1E              ADC TABASE+1
0661   85 18              STA TEMP+1
0663   A4 1F              LDY ENTLEN      ; Y-REGISTER AUF FELD FUER BELEGT/FREI - MARKE SETZEN..
0665   C8                 INY
0666   C8                 INY
0667   A9 01       LOOP   LDA #1
0669   D1 17              CMP (TEMP),Y    ; EINTRAG BELEGT ?
066B   D0 16              BNE INSERT      ; NEIN.
066D   A5 17              LDA TEMP        ; JA, VERSUCHE DEN NAECHSTEN...
066F   18                 CLC
0670   65 1F              ADC ENTLEN
0672   90 02              BCC MORE
0674   E6 18              INC TEMP+1
0676   69 03       MORE   ADC #$3         ; LABEL UEBERSPRINGEN...
0678   85 17              STA TEMP
067A   A9 00              LDA #0
067C   65 18              ADC TEMP+1
067E   85 18              STA TEMP+1
0680   4C 67 06           JMP LOOP
0683   88          INSERT DEY             ; Y-REGISTER AUF ERSTES DATENFELD ZEIGEN LASSEN...
0684   88                 DEY
0685   88          LOPE   DEY             ; OBJECT IN GEFUNDENEN EINTRAG KOPIEREN...
0686   B1 15              LDA (OBJECT),Y
0688   91 17              STA (TEMP),Y
068A   C0 00              CPY #0          ; FERTIG ?
068C   D0 F7              BNE LOPE        ; NEIN.
068E   A4 1F              LDY ENTLEN      ; JA! SETZE NUN VERKETTUNGSZEIGER IM GEFUNDENEN
0690   A5 13              LDA POINTR      ;  EINTRAG AUF DEN FOLGEEINTRAG...
0692   91 17              STA (TEMP),Y
0694   C8                 INY
0695   A5 14              LDA POINTR+1
0697   91 17              STA (TEMP),Y
0699   C8                 INY
069A   A9 01              LDA #1
069C   91 17              STA (TEMP),Y    ; SETZE MARKIERUNG AUF 'BELEGT'.
069E   A5 10              LDA INDEXD      ; MUSS VERZEICHNIS KORRIGIERT WERDEN ?
06A0   D0 0D              BNE SETINX      ; JA.
06A2   88                 DEY             ; NEIN, SETZE VERKETTUNGSZEIGER
06A3   A5 18              LDA TEMP+1      ;  DES VORANGEHENDEN EINTRAGS UM...
06A5   91 1B              STA (OLD),Y
06A7   88                 DEY
06A8   A5 17              LDA TEMP
06AA   91 1B              STA (OLD),Y
06AC   4C BB 06           JMP DONE
06AF   20 F8 06    SETINX JSR PRETAB      ; AKTUALISIERE INDLOC.
06B2   A5 17              LDA TEMP        ; BELEGE INDLOC MIT ADRESSE DES NEU EINGEFUEGTEN
06B4   91 11              STA (INDLOC),Y  ;  LISTENELEMENTS.
06B6   C8                 INY
06B7   A5 18              LDA TEMP+1
06B9   91 11              STA (INDLOC),Y
06BB   A9 FF       DONE   LDA #$FF
```

Bild 9-27: Programme zur verketteten Liste (Fortsetzung)

```
LINE #  LOC     CODE            LINE

        06BD    60              OUTE    RTS                 ; NACH ERFOLGREICHEM DURCHLAUF IST Z FLAG RUECKGESETZT.
                                ;
                                ;
                                ;
        06BE    20  00  06      DELETE  JSR SEARCH          ; SUCHE OBJECT.
        06C1    F0  34                  BEQ OUTS            ; NICHT VORHANDEN, FERTIG.
        06C3    A4  1F                  LDY ENTLEN          ; GEFUNDEN! HALTE VERKETTUNGSZEIGER
        06C5    B1  13                  LDA (POINTR),Y      ;  IN TEMP FEST...
        06C7    85  17                  STA TEMP
        06C9    C8                      INY
        06CA    B1  13                  LDA (POINTR),Y
        06CC    85  18                  STA TEMP+1
        06CE    C8                      INY
        06CF    A9  00                  LDA #0
        06D1    91  13                  STA (POINTR),Y      ; LOESCHE 'BELEGT' - MARKIERUNG.
        06D3    A5  10                  LDA INDEXD          ; IST VERZEICHNIS ZU AKTUALISIEREN ?
        06D5    F0  06                  BEQ PREINX          ; NEIN.
        06D7    20  F8  06              JSR PRETAB          ; JA! INDLOC MIT GUELTIGEM WERT BELEGEN.
        06DA    4C  EA  06              JMP MOVEIT          ; MODIFIZIERE VERZEICHNIS.
        06DD    A5  1B          PREINX  LDA OLD             ; SETZE VERKETTUNGSZEIGER DES VORANGEHENDEN
        06DF    18                      CLC                 ;  LISTENELEMENTS AUF DAS NACHFOLGENDE...
        06E0    65  1F                  ADC ENTLEN
        06E2    85  11                  STA INDLOC
        06E4    A9  00                  LDA #0
        06E6    65  1C                  ADC OLD+1
        06E8    85  12                  STA INDLOC+1
        06EA    A5  17          MOVEIT  LDA TEMP            ; GEMEINSAMER TEIL,
        06EC    A0  00                  LDY #0              ;  INDLOC WEIST AUF ZU VERAENDERNDE SPEICHERSTELLE...
        06EE    91  11                  STA (INDLOC),Y
        06F0    C8                      INY
        06F1    A5  18                  LDA TEMP+1
        06F3    91  11                  STA (INDLOC),Y
        06F5    A9  00                  LDA #0
        06F7    60              OUTS    RTS                 ; Z FLAG NACH ERFOLGREICHER ABARBEITUNG GESETZT.
                                ;
                                ;
                                ;
        06F8    A0  00          PRETAB  LDY #0              ; HINTERLEGE IN INDLOC ZU OBJECT GEHOERENDEN
        06FA    B1  15                  LDA (OBJECT),Y      ;  LISTENZEIGER AUS DEM VERZEICHNIS.
        06FC    38                      SEC                 ; ZUNAECHST ASCII - VERSCHIEBUNG ABZIEHEN...
        06FD    E9  41                  SBC #$41
        06FF    0A                      ASL A               ; ERHALTENEN WERT MIT 2 MULTIPLIZIEREN UND
        0700    18                      CLC                 ;  BASISADRESSE DES VERZEICHNISSES HINZU ADDIEREN...
        0701    65  19                  ADC REFBAS
        0703    85  11                  STA INDLOC
        0705    A9  00                  LDA #0
        0707    65  1A                  ADC REFBAS+1
        0709    85  12                  STA INDLOC+1
        070B    60                      RTS
        070C                            .END
```

Bild 9-27: Programme zur verketteten Liste (Fortsetzung)

Ein binärer Baum

Wir wollen jetzt die wichtigsten Routinen zum Umgang mit Bäumen entwickeln. Dazu legen wir die einfache Struktur von Bild 9-28 zugrunde. Es ist ein binärer Baum, dessen Verzweigungspunkte, die „Knoten“ (nodes), Namen von Personen sind. Die Namen sind intern mit Hilfe von „Etiketten“ (tags) geordnet, die die ersten drei Buchstaben des jeweiligen Namens enthalten. Im Speicher wird diese Baumstruktur wie in Bild 9-29 gezeigt wiedergegeben. Man findet dort zu jedem Knoten seinen Inhalt (den Namen) und zwei Zeiger (die von den Knoten ausgehenden Verbindungspfeile [sogenannte „links“]). Die erste Verbindung, nach links unten verzweigend, wird „linker Bruder“, die andere, nach rechts unten verzweigend, „rechter Bruder“, genannt. Der Knoten, von dem sie ausgehen, wird oft „Wurzel“ (root) genannt. So besitzt der Eintrag „Jansen“ zwei Verbindungen, „2“ und „4“. Das gibt an, daß der linke Bruder Eintrag Nummer 2 (Amann) und der rechte Bruder Eintrag Nummer 4 (Schmidt) ist. Eine „0“ im Verzweigungsfeld gibt an, daß hier kein Bruder vorhanden ist. Im Beispiel kommt das Etikett des linken Bruders alphabetisch vor dem des rechten Bruders.

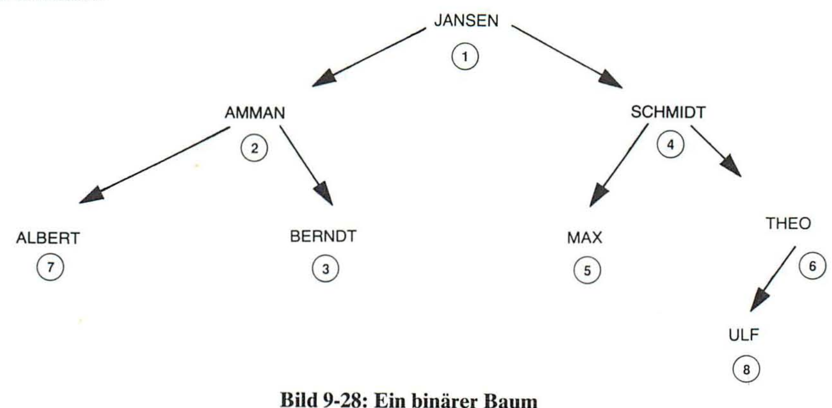


Bild 9-28: Ein binärer Baum

Man braucht zur Handhabung von Bäumen zwei Hauptroutinen, einen „Erbauer“ (tree builder) und einen „Abtaster“ (tree traverser) des Baums. Das einzufügende Element wird in einem Puffer zwischengespeichert. Den Inhalt dieses Puffers fügt der Erbauer am passenden Knoten in den Baum ein. Der Abtaster verarbeitet rekursiv die Struktur des Baums und druckt den Inhalt aller Knoten in alphabetischer Reihenfolge aus. Das Flußdiagramm für die Erbauerroutine findet sich in Bild 9-30, das für den Abtaster in Bild 9-31. Da die Abtastroutine rekursiv arbeitet, läßt sie sich nicht ohne weiteres als Flußdiagramm darstellen. Bild 9-32 gibt eine andere Beschreibung der Routine wieder. Das Format eines Knotens ist in Bild 9-33 gezeigt. Er enthält Daten der Länge ENTLEN, gefolgt von zwei 16-Bit-Zeigern (dem rechten und linken Zeiger). Um Mißverständnissen vorzubeugen: Beachten Sie, daß die Darstellung von Bild 9-29 vereinfacht ist und daß der rechte Zeiger im Speicherformat *links* vom linken Zeiger festgehalten ist. Die von unserem Programm verwendete Speicherbelegung gibt Bild 9-34 wieder und Bild 9-37 enthält das Programm.

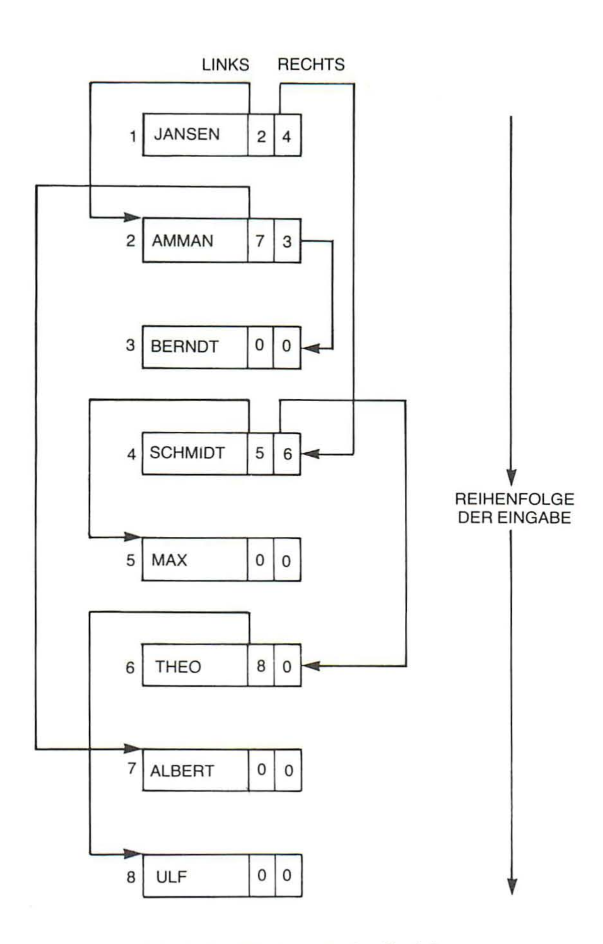


Bild 9-29: Wiedergabe im Speicher

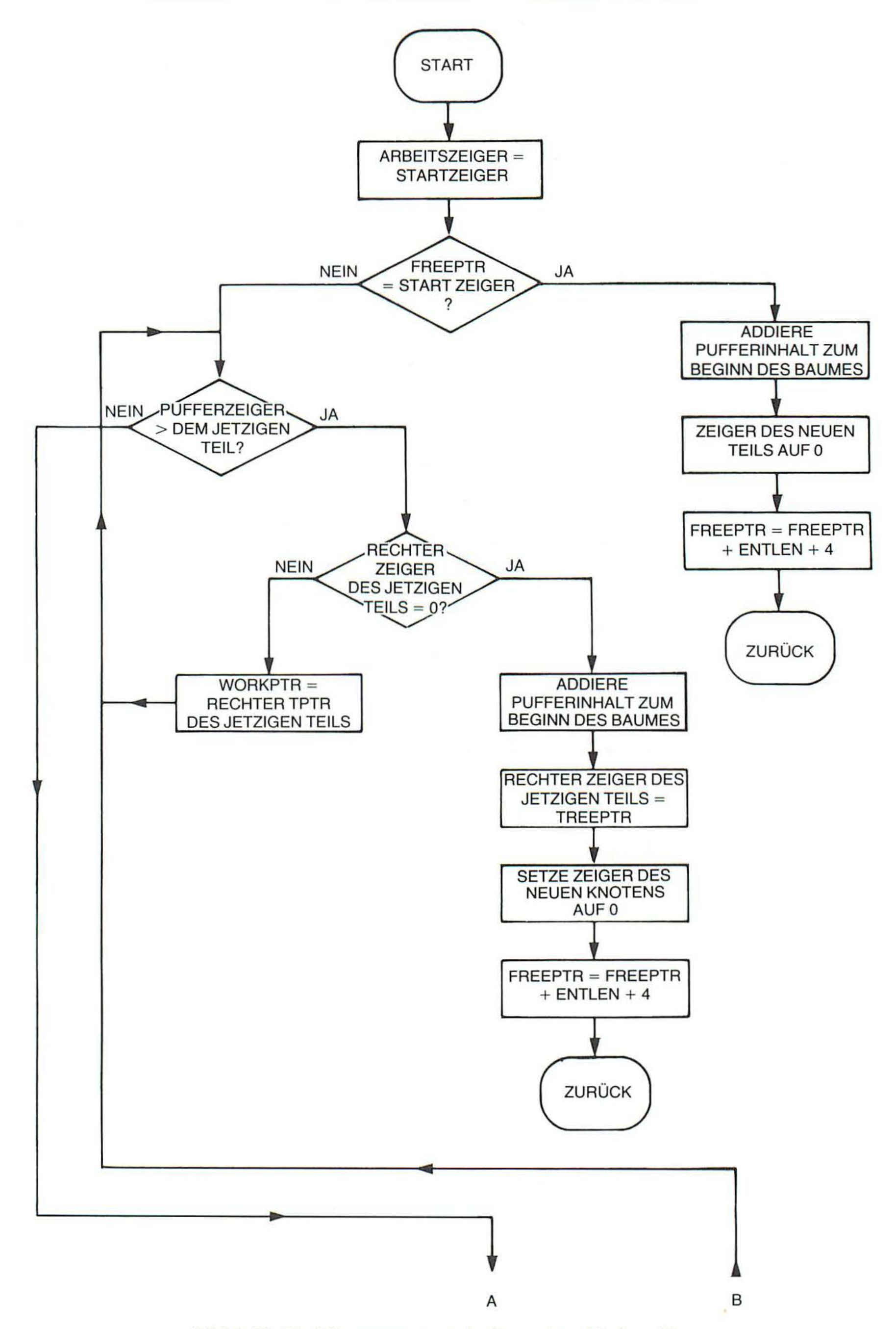


Bild 9-30: Flußdiagramm zum Aufbau eines binären Baums

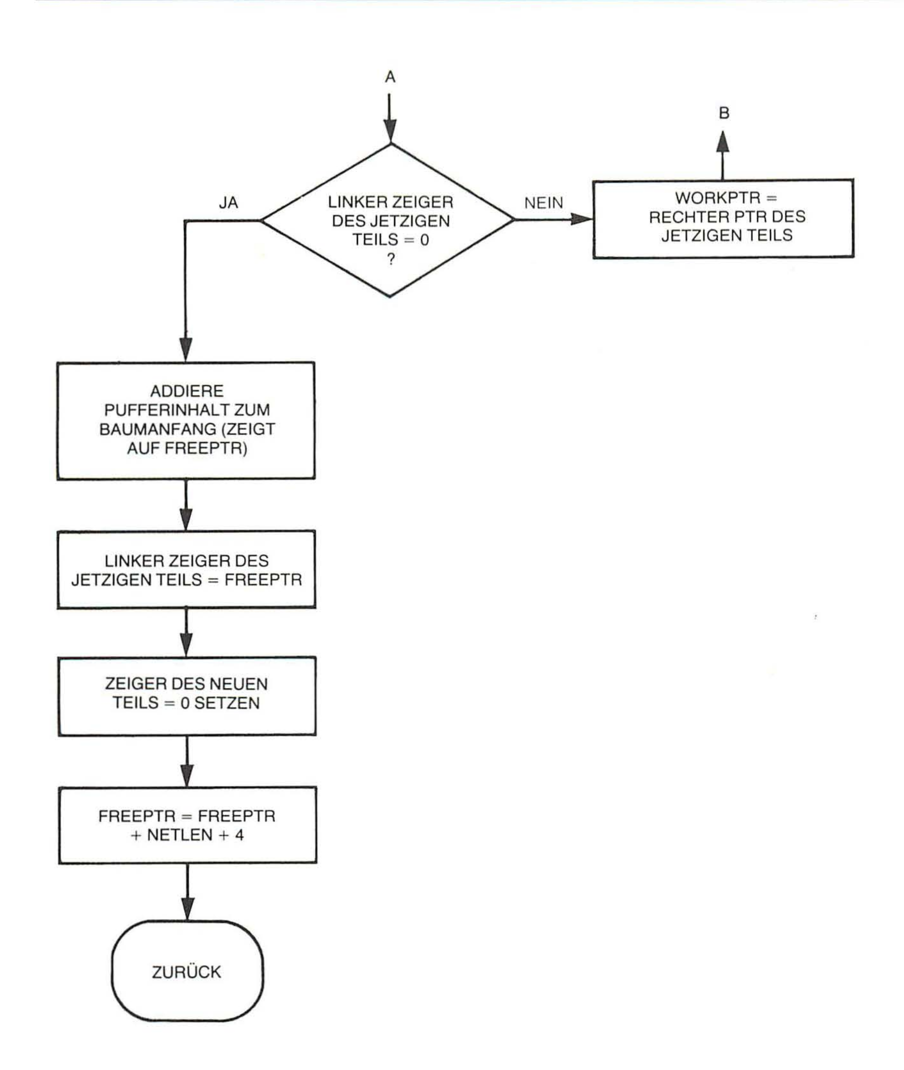


Bild 9-30: Flußdiagramm zum Aufbau eines binären Baums (Fortsetzung)

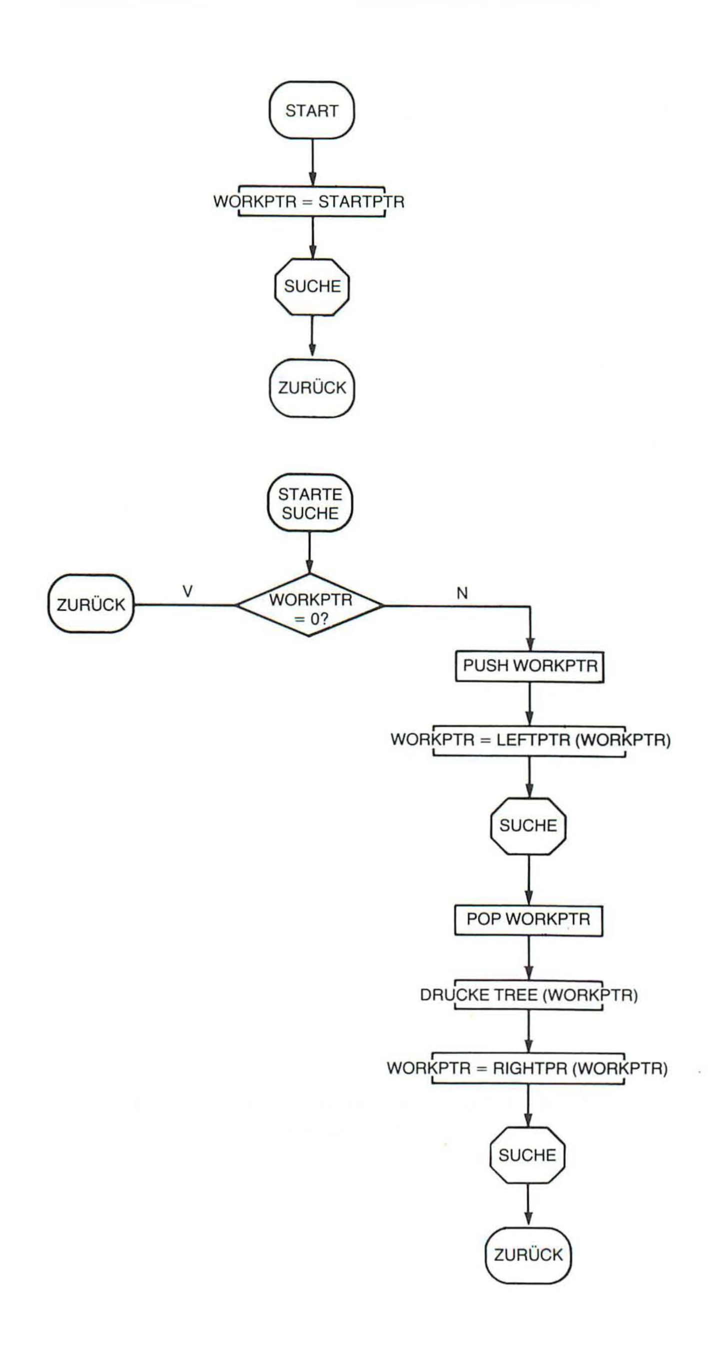


Bild 9-31: Flußdiagramm zum Absuchen eines Baums

```
PROGRAM BAUMSUCHE
BEGIN
        CALL SEARCH (STARTPOINTER);
END.

ROUTINE SUCHE (WORKPOINTER);
BEGIN
        IF WORKPOINTER = 0 THEN RETURN;
        SUCHE [LEFTPTR (WORKPOINTER)];
        PRINT TREE (WORKPOINTER);
        SUCHE [RIGHTPTR (WORKPTR)];
        RETURN;
END.
```

Bild 9-32: Algorithmus zum Absuchen eines Baums

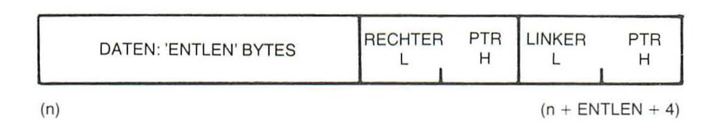


Bild 9-33: Die „Knoten": Datenelemente eines Baums

Die Einfügeroutine INSERT befindet sich dort unter den Adressen 0200 bis 0282. Das Etikett (tag) des einzufügenden Objekts wird mit dem jeweiligen Eintrag verglichen. Ist es größer, so geht man nach rechts, ist es kleiner nach links hinunter. Dieser Prozeß wird wiederholt, bis entweder eine nicht belegte Verbindung oder eine passende Einfügestelle für den neuen Knoten gefunden ist (d.h. eine Stelle, wo der eine Knoten größer, der folgende kleiner als der einzufügende ist [oder umgekehrt]). Der neue Knoten wird dann einfach durch Setzen der zugehörigen Zeiger eingebunden.
Die Abtastroutine TRAVERSE belegt die Speicheradressen 0285 bis 02C6. Die Hilfsroutinen OUT, AND und CLRPTR sind in 02C7 bis 02FE untergebracht.
Ein Beispiel einer Einfügung in einen Baum zeigt Bild 9-35, und eine Abtastung ist in Bild 9-36 wiedergegeben.

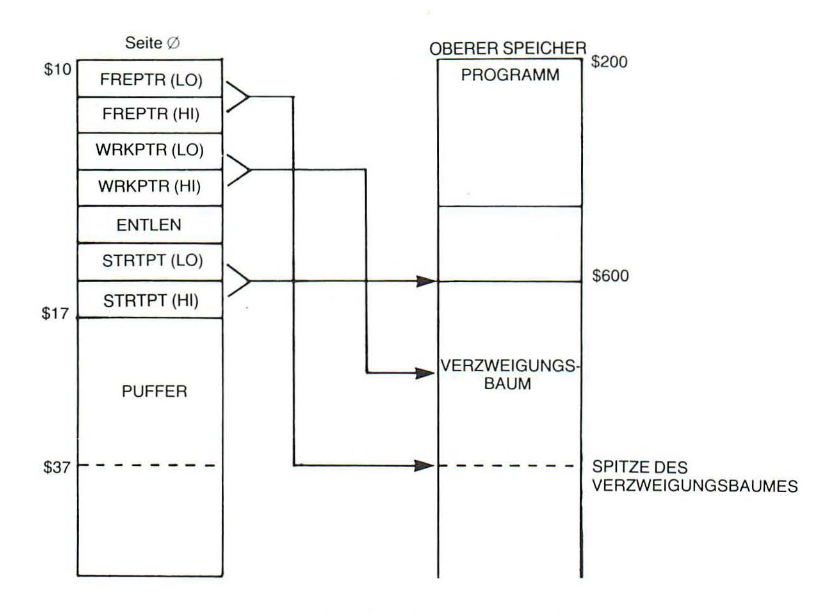


Bild 9-34: Speicherbelegung zur Baumstruktur

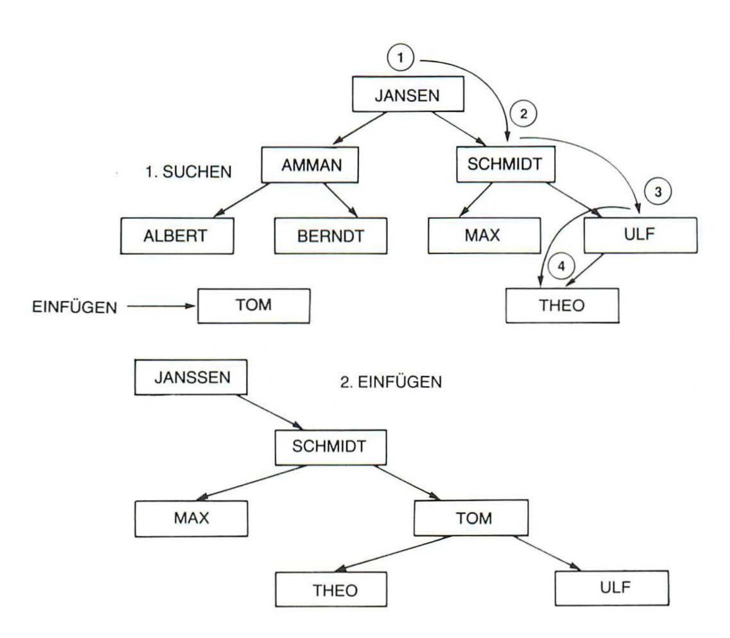


Bild 9-35: Einfügen eines Elements in einen Baum

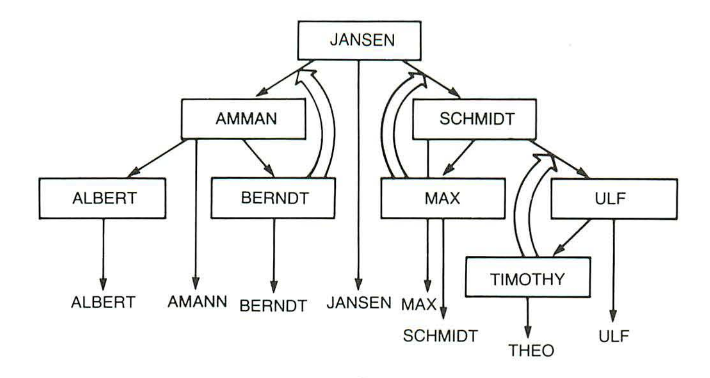


Bild 9-36: Auslisten der Baumelemente

Eine Anmerkung zu Baumstrukturen

Man kann binäre Bäume in verschiedenster Art und Weise aufbauen und abtasten. Eine andere Form unseres Baums ist z.B. in Bild 9-38 wiedergegeben. Er ist „vorsortiert" und kann so abgetastet werden:

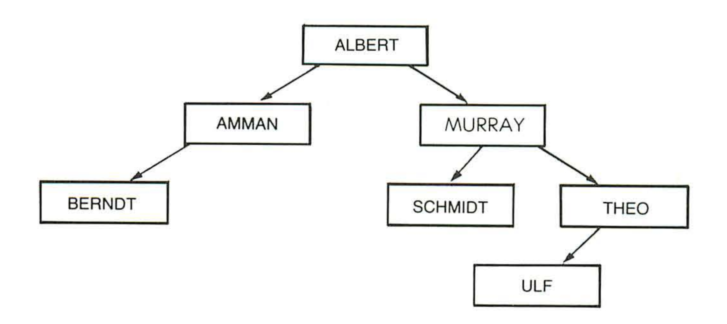


Bild 9-38: Ein vorsortierter Baum

1. Wurzel ausdrucken,
2. linken Unterbaum abtasten,
3. rechten Unterbaum abtasten.

Es gibt noch eine ganze Reihe weiterer Techniken und Vereinbarungen.

```
LOC     CODE          LINE

                      ; PROGRAMM ZUR BEARBEITUNG EINER BAUMSTRUKTUR.
                      ;
                      ; 'INSERT', DIE EINE DER BEIDEN ROUTINEN, FUEGT DEN
                      ; INHALT VON 'BUFFER' IN DEN BAUM EIN,
                      ; WAEHREND 'TRVRSE' DEN BAUM REKURSIV DURCHLAEUFT UND
                      ; DEN ALPHANUMERISCH SORTIERTEN INHALT DER KNOTEN AUSGIBT.
                      ;
                      ; HINWEIS: ES MUESSEN 'ENTLEN' INITIALISIERT UND 'FREPTR'
                      ; AUF 'STRTPT' GESETZT SEIN, BEVOR EINE DER BEIDEN
                      ; ROUTINEN AUFGERUFEN WIRD.
                      ;
0000                          * = $10
0010    00 00         FREPTR: * = * + 2       ; FREISPEICHERZEIGER, ZEIGT AUF NAECHSTEN FREIEN
0012                                          ;  SPEICHERBEREICH.
0012    00 00         WRKPTR: * = * + 2       ; ARBEITSZEIGER, WEIST AUF AKTUELLEN KNOTEN.
0014    00            ENTLEN: * = * + 1       ; LAENGE EINES EINTRAGS IM BAUM IN BYTES.
0015    00 06         STRTPT: .WORD $600      ; ZEIGER AUF DEN WURZELKNOTEN.
0017    00 00 00      BUFFER: * = * + 20      ; E/A PUFFER.
                      ;
0037                          * = $200
                      ;
                      ; EINFUEGEN EINES EINTRAGS ODER KNOTENS IN DEN BAUM.
                      ; BUFFER MUSS BEIM AUFRUF EINZUFUEGENDEN KNOTEN ENTHALTEN !
                      ;
0200    A5 15         INSERT  LDA STRTPT      ; ARBEITSZEIGER AUF WURZEL SETZEN...
0202    85 12                 STA WRKPTR
0204    A5 16                 LDA STRTPT+1
0206    85 13                 STA WRKPTR+1
0208    A5 10                 LDA FREPTR      ; WEIST FREISPEICHERZEIGER
020A    C5 15                 CMP STRTPT      ;  AUF WURZEL ?
020C    D0 0D                 BNE INLOOP      ; NEIN.
020E    A5 11                 LDA FREPTR+1
0210    C5 16                 CMP STRTPT+1
0212    D0 07                 BNE INLOOP
0214    20 D7 02              JSR ADD         ; JA, FUEGE BUFFER - INHALT AN AKTUELLER POSITION EIN.
0217    20 E4 02              JSR CLRPTR      ; SETZE ZEIGER DES AKTUELLEN KNOTENS AUF 0.
021A    60                    RTS             ; FERTIG, ERSTER KNOTEN EINGEFUEGT.
021B    A0 00         INLOOP  LDY #0          ; VERGLEICHE SCHLUESSELBEREICH DES BUFFER - INHALTS
021D    B9 17 00      CMPLP   LDA BUFFER,Y    ;  MIT DEM DES AKTUELLEN KNOTENS...
0220    D1 12                 CMP (WRKPTR),Y
0222    90 33                 BCC LESSTN      ; BUFFER-SCHLUESSEL KLEINER, HAENGE BUFFER IN LINKEN
0224                                          ;  TEILBAUM EIN.
0224    F0 02                 BEQ NXT         ; ZEICHEN SIND GLEICH, PRUEFE DIE NAECHSTEN.
0226    B0 05                 BCS GRTNEQ      ; BUFFER-SCHLUESSEL GROESSER, HAENGE BUFFER IN RECHTEN
0228                                          ;  TEILBAUM EIN.
0228    C8            NXT     INY
0229    C9 04                 CMP #$4         ; BEREITS 3 ZEICHEN VERGLICHEN ?
022B    D0 F0                 BNE CMPLP       ; NEIN, PRUEFE DIE NAECHSTEN.
022D    A4 14         GRTNEQ  LDY ENTLEN
022F    B1 12                 LDA (WRKPTR),Y  ; ZEIGER AUF RECHTEN TEILBAUM IM AKTUELLEN KNOTEN = 0 ?
0231    D0 15                 BNE NXRNOD      ; FALLS NICHT, GEHE ZUM RECHTEN TEILBAUM...
0233    C8                    INY
0234    B1 12                 LDA (WRKPTR),Y
0236    D0 10                 BNE NXRNOD
0238    A5 11                 LDA FREPTR+1    ; SETZE RECHTEN NACHFOLGERZEIGER DES AKTUELLEN KNOTENS
023A    91 12                 STA (WRKPTR),Y  ;  AUF ERSTEN FREIEN EINTRAG...
023C    88                    DEY
023D    A5 10                 LDA FREPTR
023F    91 12                 STA (WRKPTR),Y
0241    20 D7 02              JSR ADD         ; KETTE BUFFER - INHALT IN BAUM EIN.
```

Bild 9-37: Programme zum Umgang mit Baumstrukturen

```
LOC    CODE          LINE

0244   20 E4 02             JSR CLRPTR       ; SETZE NACHFOLGERZEIGER DES NEUEN KNOTENS ZURUECK.
0247   60                   RTS              ; FERTIG, RECHTER KNOTEN EINGEFUEGT.
0248   A4 14         NXRNOD LDY ENTLEN       ; SETZE ARBEITSZEIGER
024A   B1 12                LDA (WRKPTR),Y   ;  AUF RECHTEN NACHFOLGER DES
024C   AA                   TAX              ;   AKTUELLEN KNOTENS...
024D   C8                   INY
024E   B1 12                LDA (WRKPTR),Y
0250   85 13                STA WRKPTR+1
0252   86 12                STX WRKPTR
0254   4C 1B 02             JMP INLOOP       ; PRUEFE NUN DEN NEUEN AKTUELLEN KNOTEN.
0257   A4 14         LESSTN LDY ENTLEN       ; ZEIGER AUF LINKEN NACHFOLGER
0259   C8                   INY              ;  DES AKTUELLEN KNOTENS
025A   C8                   INY              ;   GLEICH 0 ?
025B   B1 12                LDA (WRKPTR),Y
025D   D0 15                BNE NXLNOD       ; NEIN, GEHE IN LINKEN TEILBAUM.
025F   C8                   INY
0260   B1 12                LDA (WRKPTR),Y
0262   D0 10                BNE NXLNOD
0264   A5 11                LDA FREPTR+1     ; JA, SETZE LINKEN NACHFOLGERZEIGER DES
0266   91 12                STA (WRKPTR),Y   ;  AKTUELLEN KNOTENS AUF ERSTEN FREIEN EINTRAG...
0268   88                   DEY
0269   A5 10                LDA FREPTR
026B   91 12                STA (WRKPTR),Y
026D   20 D7 02             JSR ADD          ; BUFFER - INHALT IN BAUM EINKETTEN.
0270   20 E4 02             JSR CLRPTR       ; NACHFOLGERZEIGER DES NEUEN KNOTENS RUECKSETZEN.
0273   60                   RTS              ; FERTIG, BUFFER-INHALT IN LINKEN TEILBAUM EINGEKETTET.
0274   A4 14         NXLNOD LDY ENTLEN       ; SETZE ARBEITSZEIGER
0276   C8                   INY              ;  AUF LINKEN NACHFOLGER
0277   C8                   INY              ;   DES AKTUELLEN KNOTENS...
0278   B1 12                LDA (WRKPTR),Y
027A   AA                   TAX
027B   C8                   INY
027C   B1 12                LDA (WRKPTR),Y
027E   85 13                STA WRKPTR+1
0280   86 12                STX WRKPTR
0282   4C 1B 02             JMP INLOOP       ; NEUEN AKTUELLEN KNOTEN PRUEFEN.
                     ;
                     ; BAUMDURCHLAUF:  LISTET DIE KNOTENINHALTE IN ALPHANUMERISCHER
                     ; REIHENFOLGE AUF.
                     ; ZUR AUSGABE DES PUFFERS AUF EIN GERAET WIRD EINE ENTSPRECHENDE
                     ; DIENSTROUTINE BENOETIGT.
                     ;
0285   A5 15         TRVRSE LDA STRTPT       ; SETZE ARBEITSZEIGER AUF WURZELKNOTEN...
0287   85 12                STA WRKPTR
0289   A5 16                LDA STRTPT+1
028B   85 13                STA WRKPTR+1
028D   A5 13         SEARCH LDA WRKPTR+1
028F   A6 12                LDX WRKPTR       ; FALLS ARBEITSZEIGER UNGLEICH 0, WEITER,
0291   D0 07                BNE OK
0293   A4 13                LDY WRKPTR+1
0295   D0 03                BNE OK
0297   4C C6 02             JMP RETN         ; SONST RUECKSPRUNG !
029A   48            OK     PHA              ; ARBEITSZEIGER
029B   8A                   TXA              ;  AUF STACK
029C   48                   PHA              ;   ABLEGEN.
029D   A4 14                LDY ENTLEN       ; ARBEITSZEIGER AUF LINKEN NACHFOLGER
029F   C8                   INY              ;  DES AKTUELLEN KNOTENS
02A0   C8                   INY              ;   SETZEN...
02A1   B1 12                LDA (WRKPTR),Y
02A3   AA                   TAX
```

Bild 9-37: Programme zum Umgang mit Baumstrukturen (Fortsetzung)

```
LOC     CODE            LINE

02A4    C8                      INY
02A5    B1  12                  LDA (WRKPTR),Y
02A7    85  13                  STA WRKPTR+1
02A9    86  12                  STX WRKPTR
02AB    20  8D  02              JSR SEARCH      ; SUCHE REKURSIV FORTSETZEN.
02AE    68                      PLA             ; ZEIGER AUF EHEMALIGEN AKTUELLEN KNOTEN
02AF    85  12                  STA WRKPTR      ;  VOM STACK HOLEN...
02B1    68                      PLA
02B2    85  13                  STA WRKPTR+1
02B4    20  C7  02              JSR OUT         ; INHALT DES AKTUELLEN KNOTENS AUSGEBEN.
02B7    A4  14                  LDY ENTLEN      ; ARBEITSZEIGER AUF RECHTEN
02B9    B1  12                  LDA (WRKPTR),Y  ;  NACHFOLGERZEIGER DES AKTUELLEN
02BB    AA                      TAX             ;   KNOTENS SETZEN...
02BC    C8                      INY
02BD    B1  12                  LDA (WRKPTR),Y
02BF    85  13                  STA WRKPTR+1
02C1    86  12                  STX WRKPTR
02C3    20  8D  02              JSR SEARCH      ; RECHTEN TEILBAUM ABARBEITEN.
02C6    60              RETN    RTS             ; FERTIG !
                        ;
                        ; TRANSFER DES INHALTS DES AKTUELLEN KNOTENS IN 'BUFFER',
                        ; AUSGABE DESSELBEN.
                        ;
02C7    A0  00          OUT     LDY #0
02C9    B1  12          XFR     LDA (WRKPTR),Y  ; ZEICHEN AUS AKTUELLEM KNOTEN HOLEN
02CB    99  17  00              STA BUFFER,Y    ;  UND IN BUFFER ABSPEICHERN.
02CE    C8                      INY
02CF    C4  14                  CPY ENTLEN      ; ALLE ZEICHEN KOPIERT ?
02D1    D0  F6                  BNE XFR         ; NEIN, WEITER.
02D3    EA                      NOP             ; JA! (HIER AUFRUF DER
02D4    EA                      NOP             ;  AUSGABEROUTINE
02D5    EA                      NOP             ;   EINSETZEN !)
02D6    60                      RTS             ; FERTIG.
                        ;
                        ; UBERTRAGUNG DES INHALTS VON 'BUFFER' IN DEN EINZUFUEGENDEN
                        ; FREIEN EINTRAG.
                        ;
02D7    A0  00          ADD     LDY #0
02D9    B9  17  00      MOV     LDA BUFFER,Y    ; ZEICHEN AUS BUFFER HOLEN
02DC    91  10                  STA (FREPTR),Y  ;  UND IM FREIEN KNOTEN ABLEGEN.
02DE    C8                      INY
02DF    C4  14                  CPY ENTLEN      ; ALLE ZEICHEN KOPIERT ?
02E1    D0  F6                  BNE MOV         ; NEIN, WEITER.
02E3    60                      RTS             ; JA, FERTIG.
                        ;
                        ; NACHFOLGERZEIGER DES NEUEN KNOTENS LOESCHEN UND
                        ; FREISPEICHERZEIGER AKTUALISIEREN.
                        ;
02E4    A4  14          CLRPTR  LDY ENTLEN
02E6    A9  00                  LDA #0
02E8    A2  04                  LDX #$4         ; 4 SCHLEIFENDURCHLAEUFE !
02EA    91  10          CLRLP   STA (FREPTR),Y  ; ZEIGERBYTE LOESCHEN.
02EC    C8                      INY
02ED    CA                      DEX             ; FERTIG ?
02EE    D0  FA                  BNE CLRLP       ; NEIN, WEITER !
02F0    A5  14                  LDA ENTLEN      ; ZUR AKTUALISIERUNG DES FREISPEICHERZEIGERS
02F2    18                      CLC             ;  LAENGE EINES EINTRAGS PLUS
02F3    69  04                  ADC #$4         ;   4 BYTES FUER DIE ZEIGER
02F5    65  10                  ADC FREPTR      ;    AUF ALTEN ZEIGERWERT ADDIEREN.
02F7    90  02                  BCC CC
```

```
LOC     CODE            LINE

02F9    E6  11                  INC FREPTR+1    ; UEBERTRAG.
02FB    85  10          CC      STA FREPTR      ; FREISPEICHERZEIGER NEU BELEGEN.
02FD    60                      RTS             ; FERTIG!
02FE                            .END
```

Bild 9-37: Programme zum Umgang mit Baumstrukturen (Fortsetzung)

Ein Hashing-Algorithmus

Ein verbreitetes Problem beim Entwurf von Datenstrukturen ist die Frage, wie man in einem beschränkten Speicherumfang die Namen und Daten systematisch so ablegt, daß man sie später einfach wiedergewinnen kann. Falls es sich nicht ausnahmsweise um lauter verschiedene Zahlen (oder Lücken dazwischen) handelt, gibt es kaum einen Weg, den Speicher lückenlos und doch systematisch zu belegen. Insbesondere gilt das für Namen: Will man sie so im Speicher ablegen, daß man sie ohne viel Aufwand wiedergewinnen kann (d.h. in alphabetischer Reihenfolge) und sollen beim Abspeichern Schiebeoperationen möglichst unterbleiben, so bräuchte man einen gewaltigen Speicheraufwand, da für jeden möglichen Namen ein Platz reserviert werden müßte. Das ist völlig unannehmbar. Die Lösung des Problems liegt in der Verwendung eines besonderen Hashing-Algorithmus, der jedem Namen im Speicher eine eindeutige (oder nahezu eindeutige) Nummer zuweist. (Hashing bezeichnet das feine Zerkleinern eines Gegenstands. Ein Hashing-Algorithmus teilt entsprechend den verfügbaren Speicherraum in viele kleine, möglichst gleichmäßig verteilte Stücke auf.) Die hierzu verwendete mathematische Funktion soll einfach genug sein, damit der Algorithmus schnell arbeitet, andererseits aber auch ausgefeilt genug, um eine möglichst gleichmäßige Speicherbelegung zu erreichen. Man kann die sich aus dem Algorithmus ergebende Zahl als Indexzeiger verwenden und so rasch auf die jeweilige Information zugreifen.

Da es keinen Algorithmus gibt, der garantiert, daß keine zwei Namen denselben Speicherplatz zugewiesen bekommen (es entsteht dann eine „Kollision"), braucht man eine Technik, derartige Kollisionen zu umgehen. Ein guter Hashing-Algorithmus verteilt die Namen gleichmäßig über den verfügbaren Speicherraum und erlaubt einen raschen Zugriff auf die festgehaltene Information. Der hier benutzte Hashing-Algorithmus ist sehr einfach: Wir verknüpfen einfach alle Bytes des Schlüssels durch EXKLUSIV-ODER. Um die Verteilung zu verbessern, wird nach jeder Verknüpfung eine Rotation des Bytes durchgeführt.

Kollisionen werden ganz einfach sequentiell gelöst: Man verwendet schlichtweg den nächsten verfügbaren Block in der Tabelle für den Eintrag. Das läßt sich mit einem Notizbuch vergleichen. Nehmen wir an, wir wollten „Schmitz" dort eintragen. Nun ist aber die Adreßseite mit „S" schon voll, also nehmen wir notgedrungen die nächste Seite („T") zu diesem Zweck. Das muß nicht unbedingt eine Kollision mit dem nächsten „T"-Eintrag ergeben. Wir können z. B. den „S"-Eintrag dort wieder löschen, bevor ein solcher „T"-Eintrag notwendig wird.

Beachten Sie auch, daß es eine ganze Kollisionskette geben kann. Wenn diese Kette lang, die Tabelle aber nicht voll ist, dann ist der Hashing-Algorithmus in der Regel für den gegebenen Zweck nicht sehr brauchbar. (Es gibt keinen universellen Hashing-Algorithmus!)

Da es bequem ist, bei einem Computer dem Datenformat eine Länge mit einer Zweierpotenz zu geben, wird hier eine Länge von acht Zeichen angenommen: Sechs von ihnen sind dem Schlüsselwort zugeordnet, zwei den Daten. Das ist eine typische Situation, wenn man – beispielsweise für einen Assembler – eine Symboltabelle erstellen muß. Man erlaubt hier für das Symbol selbst sechs alphanumeriche Zeichen und weist den zwei verbleibenden Bytes die Adresse zu, die das Symbol bezeichnet.

Die zum Wiedergewinnen der Information aufzuwendende Zeit ist von der Größe der Tabelle unabhängig, hängt aber vom Belegungsgrad ab. Üblicherweise gibt eine Belegung unter 80 Prozent einen raschen Zugriff (ein oder zwei Versuche). In unserem Beispiel muß sich das aufrufende Programm um die Belegungsdichte der Tabelle kümmern und vor allem verhindern, daß sie überläuft.

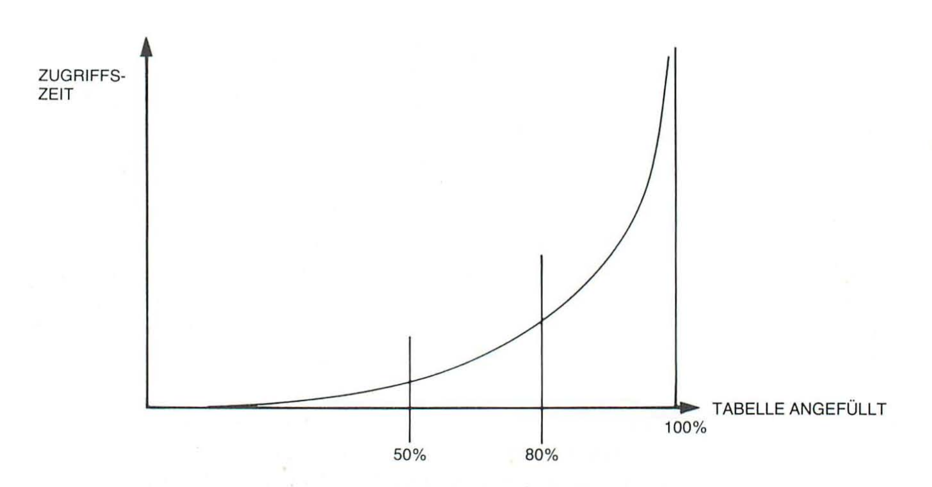


Bild 9-39: Verhalten der Zugriffszeit bei steigender Belegungsdichte des Speicherbereichs in einem Hashing-Algorithmus

Der Anstieg der Zugriffszeit bei wachsender Tabellenbelegung wird durch Bild 9-39 verdeutlicht. Bilder 9-40 und 9-43 geben die Flußdiagramme für die folgenden Teilroutinen wieder: Initialisierung (9-40), Speichern (9-41), Wiedergewinnen (9-42) und Hash (9-43). Die Speicherbelegung findet sich in Bild 9-44 und das Programm selbst in Bild 9-45. Das Programm soll alle Grundalgorithmen für einen Hashing-Mechanismus verdeutlichen. Sollen diese Routinen in eine Praxisanwendung eingebettet werden, so sind allerdings unbedingt die notwendigen Verwaltungsroutinen dazuzugeben, damit unerwartete Situationen ausgeschlossen werden können. Insbesondere muß man sicherstellen, daß die Tabelle nicht voll wird oder daß ein unzulässiger Schlüssel verwendet wird, da sonst das Programm in eine endlose Schleife geraten kann. Sie sollten das Programm sehr sorgfältig studieren. Es entmystifiziert nicht nur die Hash-Technik, sondern hilft auch ein wichtiges praktisches Problem beim Assemblerentwurf oder ähnlichen Aufgaben zu lösen, wo rasch auf Namenstabellen zugegriffen werden muß.

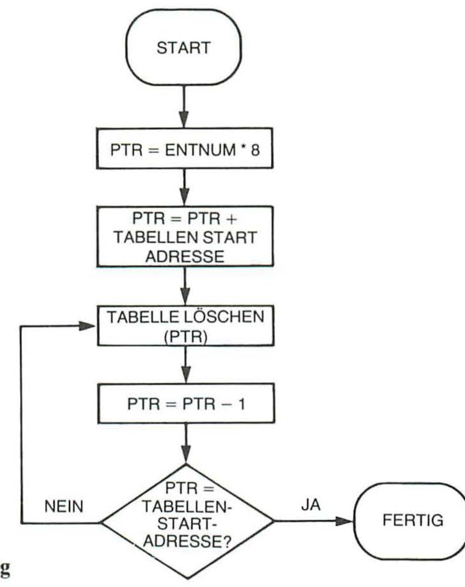


Bild 9-40: Unterprogramm zur Initialisierung

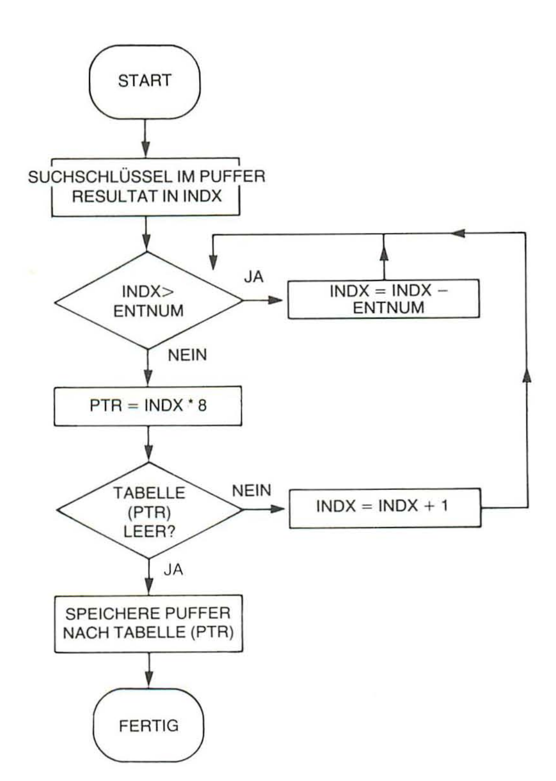


Bild 9-41: Speicherroutine „Store“

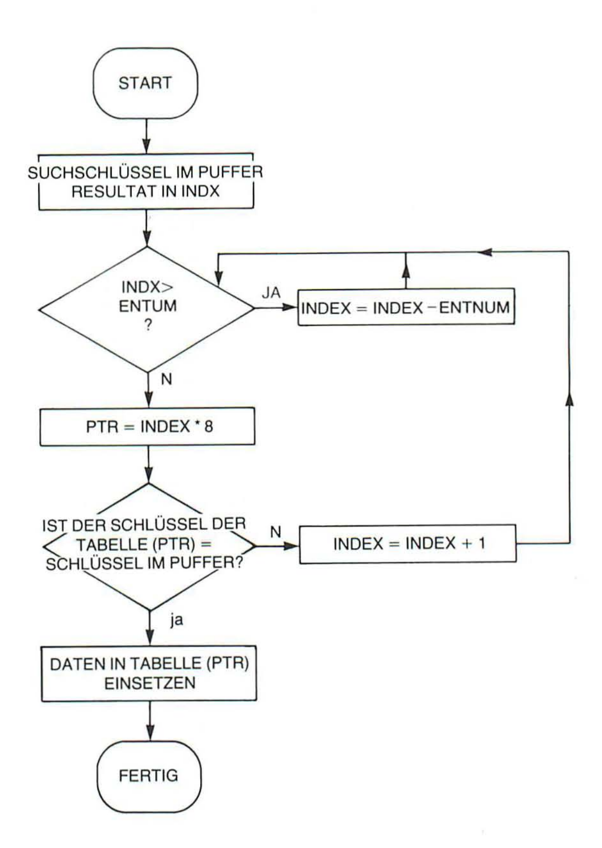


Bild 9-42: Wiedergewinnungsroutine „Retrieve“

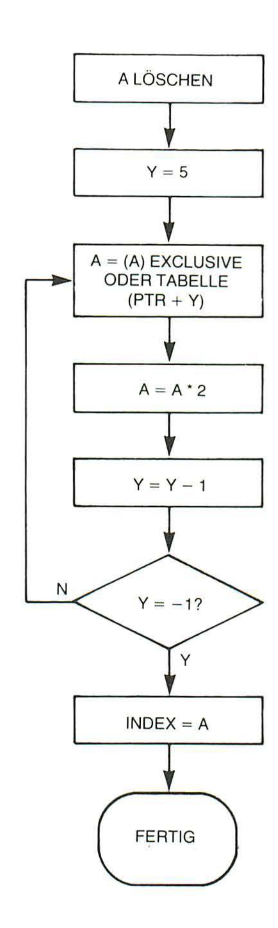


Bild 9-43: Hash-Routine

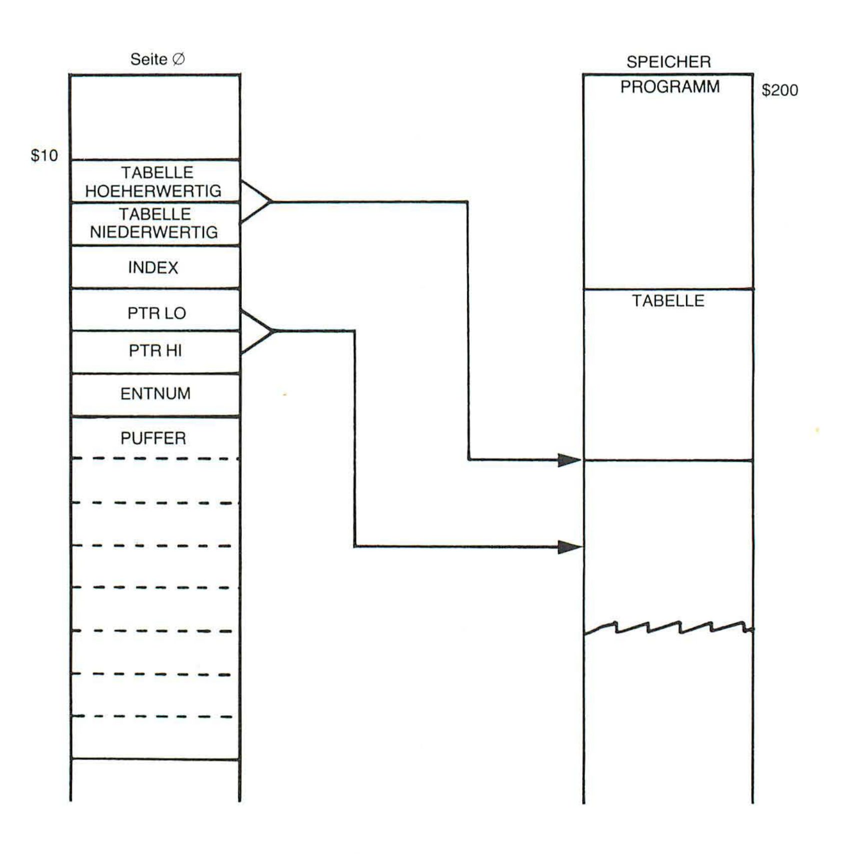


Bild 9-44: Speichern und Wiedergewinnen mit der Hash-Methode: Speicherbelegung

```
LOC     CODE          LINE

                      ; ABSPEICHERUNG VON ASSEMBLERSYMBOLEN IN EINER TABELLE,
                      ; DER ZUGRIFF ERFOLGT MIT HILFE EINES HASHING-ALGORITHMUS.
                      ; EIN EINTRAG BESTEHT AUS 6 ZEICHEN- UND 2 DATEN-BYTES,
                      ; 'ENTNUM' SOLLTE DIE MAXIMALE ANZAHL DIESER 8 BYTE LANGEN
                      ; TABELLENEINTRAEGE ENTHALTEN,
                      ; DIE ANFANGSADRESSE DER TABELLE SOLLTE IN 'TABLE'
                      ; HINTERLEGT SEIN.
                      ; HINWEIS:  DIE TABELLE MUSS VOR GEBRAUCH DURCH DIE ROUTINE
                      ; 'INIT' INITIALISIERT WERDEN.
                      ; DIE EINHALTUNG DER TABELLENGROESSE LIEGT IN DER VERANTWORTUNG
                      ; DES AUFRUFENDEN PROGRAMMS.
                      ;
0000                         * = $10
0010    00 06         TABLE  .WORD $600     ; ANFANGSADRESSE DER TABELLE.
0012    00            INDX   * = * + 1      ; NUMMER DES GEWUENSCHTEN EINTRAGS.
0013    00 00         PTR    * = * + 2      ; ZEIGER AUF TABELLENEINTRAG.
0015    00            ENTNUM * = * + 1      ; ANZAHL DER TABELLENEINTRAEGE (MAX. 256).
0016    00 00 00      BUFFER * = * + 8      ; E/A PUFFER.
                      ;
001E                         * = $200
                      ;
                      ; 'INIT':  INITIALISIEREN DER TABELLE MIT NULLEN.
                      ;
0200    A5 15         INIT   LDA ENTNUM
0202    85 13                STA PTR        ; ANZAHL DER EINTRAEGE IN POINTR ABLEGEN.
0204    20 72 02             JSR SHADD      ; ZEIGER AUF TABELLENENDE SETZEN.
0207    A2 00                LDX #0         ; INDEXREGISTER LOESCHEN.
0209    A9 00         CLRLP  LDA #0         ; KONSTANTE ZUR VORBESETZUNG DER TABELLE.
020B    A4 13                LDY PTR
020D    D0 02                BNE DECR       ; NIEDERWERTIGES BYTE DES ZEIGERS UNGLEICH NULL,
020F                                        ;  HOEHERWERTIGES BYTE NICHT HERUNTERZAEHLEN !
020F    C6 14                DEC PTR+1      ; HOEHERWERTIGES BYTE DES ZEIGERS HERUNTERZAEHLEN.
0211    C6 13         DECR   DEC PTR        ; DTO. NIEDERWERTIGES BYTE.
0213    81 13                STA (PTR,X)    ; SPEICHERZELLE LOESCHEN.
0215    A5 13                LDA PTR
0217    C5 10                CMP TABLE      ; AM TABELLENANFANG ANGEKOMMEN ?
0219    D0 EE                BNE CLRLP      ; NEIN, WEITER.
021B    A5 14                LDA PTR+1      ; DIE JEWEILS NIEDERWERTIGEN BYTES SIND GLEICH,
021D    C5 11                CMP TABLE+1    ;  STIMMEN DIE HOEHERWERTIGEN UEBEREIN ?
021F    D0 E8                BNE CLRLP      ; NEIN, WEITER.
0221    60                   RTS            ; JA, FERTIG !
                      ;
                      ; 'STORE':  ABSPEICHERN DES INHALTS VON 'BUFFER' IN DER TABELLE.
                      ; DIE ERSTEN 6 ZEICHEN DES PUFFERINHALTS WERDEN BENUTZT, UM DURCH
                      ; HASHING DIE PASSENDE TABELLENADRESSE ZU BESTIMMEN.
                      ;
0222    A2 00         STORE  LDX #0         ; INDEXREGISTER LOESCHEN.
0224    20 90 02             JSR HASH       ; BESTIMME TABELLENADRESSE.
0227    20 62 02      CMPR1  JSR LIMIT      ; ADRESSE MUSS INNERHALB DER TABELLENGRENZEN LIEGEN !
022A    A1 13                LDA (PTR,X)    ; IST DER AUSGEWAEHLTE TABELLENEINTRAG LEER ?
022C    F0 05                BEQ EMPTY      ; JA !
022E    E6 12                INC INDX       ; NEIN, KOLLISION! VERSUCHE NAECHSTEN EINTRAG...
0230    4C 27 02             JMP CMPR1
0233    A0 07         EMPTY  LDY #$7        ; 8 SCHLEIFENDURCHLAEUFE...
0235    B9 16 00      FILL   LDA BUFFER,Y   ; EIN BYTE AUS BUFFER
0238    91 13                STA (PTR),Y    ;  IN TABELLENEINTRAG KOPIEREN.
023A    88                   DEY            ; FERTIG ?
023B    10 F8                BPL FILL       ; NEIN, WEITER.
023D    60                   RTS            ; JA, ELEMENT EINGEFUEGT.
                      ;
```

Bild 9-45: Speichern und Wiedergewinnen mit der Hash-Methode: Das Programm

```
LOC    CODE         LINE

                    ; 'FIND': SUCHT EINEN TABELLENEINTRAG, DESSEN SCHLUESSEL MIT DEM IN
                    ; 'BUFFER' STEHENDEN UEBEREINSTIMMT.
                    ; DER EINTRAG WIRD, FALLS AUFFINDBAR, EINSCHLIESSLICH SEINER 2 DATEN-
                    ; BYTES IN 'BUFFER' KOPIERT.
                    ;
023E   A2 00        FIND    LDX #0        ; INDEXREGISTER LOESCHEN.
0240   20 90 02             JSR HASH      ; ADRESSE GEMAESS HASHING-ALGORITHMUS BERECHNEN.
0243   20 62 02     CMPR2   JSR LIMIT     ; ADRESSE MUSS INNERHALB DER TABELLENGRENZEN LIEGEN !
0246   A0 05                LDY #$5       ; 6 SCHLEIFENDURCHLAEUFE...
0248   B1 13        CHKLP   LDA (PTR),Y   ; ZEICHEN IN TABELLENEINTRAG
024A   D9 16 00             CMP BUFFER,Y  ;  MIT DEM ENTSPRECHENDEN IN BUFFER IDENTISCH ?
024D   D0 0E                BNE BAD       ; NEIN, PRUEFE NAECHSTEN TABELLENEINTRAG.
024F   88                   DEY           ; ALLE 6 ZEICHEN UEBERPRUEFT ?
0250   10 F6                BPL CHKLP     ; NEIN, WEITER.
0252   A0 07        MATCH   LDY #$7       ; JA, NUN 8 SCHLEIFENDURCHLAEUFE UM DATEN
0254   B1 13        XFER    LDA (PTR),Y   ;  AUS TABELLENEINTRAG
0256   99 16 00             STA BUFFER,Y  ;   IN PUFFER ZU UEBERTRAGEN.
0259   88                   DEY           ; FERTIG ?
025A   10 F8                BPL XFER      ; NEIN, WEITER.
025C   60                   RTS           ; JA. RUECKSPRUNG.
025D   E6 12        BAD     INC INDX      ; EINTRAG NICHT GEFUNDEN, VERSUCHE DEN NAECHSTEN...
025F   4C 43 02             JMP CMPR2
                    ;
                    ; 'LIMIT':  ZUNAECHST WIRD SICHERGESTELLT, DASS DER INDEX INNERHALB DER
                    ; DURCH 'ENTNUM' DEFINIERTEN TABELLENGRENZEN LIEGT.
                    ; ANSCHLIESSEND WIRD DER INDEX MIT 8 MULTIPLIZIERT UND DIE ANFANGS-
                    ; ADRESSE DER TABELLE HINZU ADDIERT.
                    ; DAS ERGEBNIS IST EIN ZEIGER AUF EINEN TABELLENEINTRAG, DER IN 'PTR'
                    ; HINTERLEGT WIRD.
                    ;
0262   A5 12        LIMIT   LDA INDX
0264   C5 15        TEST    CMP ENTNUM    ; INDX > MAX. EINTRAGSZAHL ?
0266   90 06                BCC OK        ; NEIN, OK.
0268   38                   SEC           ; JA.
0269   E5 15                SBC ENTNUM    ; ENTNUM SOLANGE SUBTRAHIEREN, BIS INDX ZULAESSIG !
026B   4C 64 02             JMP TEST
026E   85 13        OK      STA PTR       ; GUELTIGEN INDEX IN POINTR FESTHALTEN.
0270   85 12                STA INDX      ; NEUEN INDEX SICHERN.
0272   A9 00        SHADD   LDA #0        ; HOEHERWERTIGES BYTE DES ZEIGERS FUER SHIFT LOESCHEN.
0274   85 14                STA PTR+1
0276   06 13                ASL PTR       ; PTR 3 MAL LINKS SHIFT = MULTIPLIKATION MIT 8 ...
0278   26 14                ROL PTR+1
027A   06 13                ASL PTR
027C   26 14                ROL PTR+1
027E   06 13                ASL PTR
0280   26 14                ROL PTR+1
0282   18                   CLC
0283   A5 10                LDA TABLE     ; ANFANGSADRESSE DER TABELLE ADDIEREN
0285   65 13                ADC PTR       ;  UND IN PTR ABSPEICHERN...
0287   85 13                STA PTR
0289   A5 11                LDA TABLE+1
028B   65 14                ADC PTR+1
028D   85 14                STA PTR+1
028F   60                   RTS           ; FERTIG.
                    ;
                    ; BERECHNUNG DER ADRESSE EINES EINTRAGS IN DER TABELLE ANHAND DES
                    ; SCHLUESSELS DURCH HASHING.
                    ;
0290   A9 00        HASH    LDA #0        ; INDEXREGISTER LOESCHEN.
0292   18                   CLC           ; ADDITION VORBEREITEN.
```

```
LOC    CODE         LINE

0293   A0 05                LDY #$5       ; SCHLUESSELLAENGE = 6, ALSO 6 SCHLEIFENDURCHLAEUFE !
0295   59 16 00     EXOR    EOR BUFFER,Y  ; AKKU = AKKU (EXOR) PUFFERZEICHEN.
0298   2A                   ROL A         ; AKKU MIT 2 MULTIPLIZIEREN.
0299   88                   DEY           ; FERTIG ?
029A   10 F9                BPL EXOR      ; NEIN, WEITER.
029C   85 12                STA INDX      ; ERGEBNIS ALS TABELLENINDEX ABSPEICHERN.
029E   60                   RTS           ; FERTIG.
029F                        .END
```

Bild 9-45: Speichern und Wiedergewinnen mit der Hash-Methode: Das Programm (Fortsetzung)

Der Bubble-Sort

Unter „Bubble-Sort" versteht man eine Sortiertechnik, bei der die Elemente einer Tabelle in auf- oder absteigender Folge umgeordnet werden. Sie hat ihren Namen („Blasen"-Sortierung – „Bubble-Sort" ist jedoch zum stehenden Begriff geworden) von der Tatsache, daß (bei aufsteigender Folge), das kleinste Element wie eine Blase nach oben steigt. Jedesmal, wenn es mit einem größeren Element zusammenstößt, wandert es einfach um es herum.
Ein praktisches Beispiel für eine Bubble-Sortierung zeigt Bild 9-46. Die zu sortierende Liste enthält die Elemente 10, 5, 0, 2 und 100 und soll in aufsteigender Folge („0" steht vorne in der Tabelle) geordnet werden. Der Algorithmus ist einfach. Das zugehörige Flußdiagramm zeigt Bild 9-47.
Die beiden obersten (oder untersten) Elemente werden verglichen. Wenn das untere kleiner („leichter") als das obere ist, so werden sie ausgetauscht. Andernfalls bleiben sie unverändert. Aus praktischen Gründen merkt man sich eine etwa eingetretene Vertauschung und geht dann zum nächsten Element über, bis alle Elemente miteinander verglichen worden sind.
Dieser erste Durchlauf wird in den Schritten 1 bis 6 in Bild 9-47 vollzogen, wo von unten nach oben vorgegangen wird. (Wir könnten genausogut auch von oben nach unten arbeiten.)
Wenn keine Elemente mehr ausgetauscht worden sind, ist die Sortierung beendet. War etwas auszutauschen, so wird ein neuer Durchgang nötig.
In Bild 9-47 waren z.B. vier Durchgänge durch die Liste erforderlich.
Der eben beschriebene Vorgang ist einfach und wird oft eingesetzt.
Es gibt allerdings eine Schwierigkeit im Vertauschungsmechanismus. Wenn wir A und B vertauschen, so können wir nicht schreiben:

A = B
B = A

denn dann wäre der Wert von A verlorengegangen. (Probieren Sie es an einem Beispiel aus.)

Das richtige Vorgehen benutzt eine Hilfsvariable (z.B. TEMP), um den Wert von A während der Vertauschung zu retten:

TEMP = A
A = B
B = TEMP

Das funktioniert. (Versuchen Sie auch hier ein Beispiel.) Man nennt dies eine zyklische Vertauschung und es ist genau das, was das Programm macht. Die Technik ist im Flußdiagramm in Bild 9-47 wiedergegeben.

Bild 9.48 zeigt die Speicherbelegung für das „Bubble-Sort"-Programm. Hier ist jedes Element eine positive 8-Bit-Zahl. Das Programm selbst beginnt mit Adresse 200. Register X dient zum Merken von Vertauschungen, Register Y ist der Arbeitszeiger für die Tabelle. TAB soll die Anfangsadresse der zu bearbeitenden Tabelle sein. Sie finden das Programm in Bild 9.49. Es benutzt durchgängig indirekt-indizierte Adressierung. Beachten Sie die Kürze des Programms, die aus der Leistungsfähigkeit der indirekten Adressierung beim 6502 resultiert.

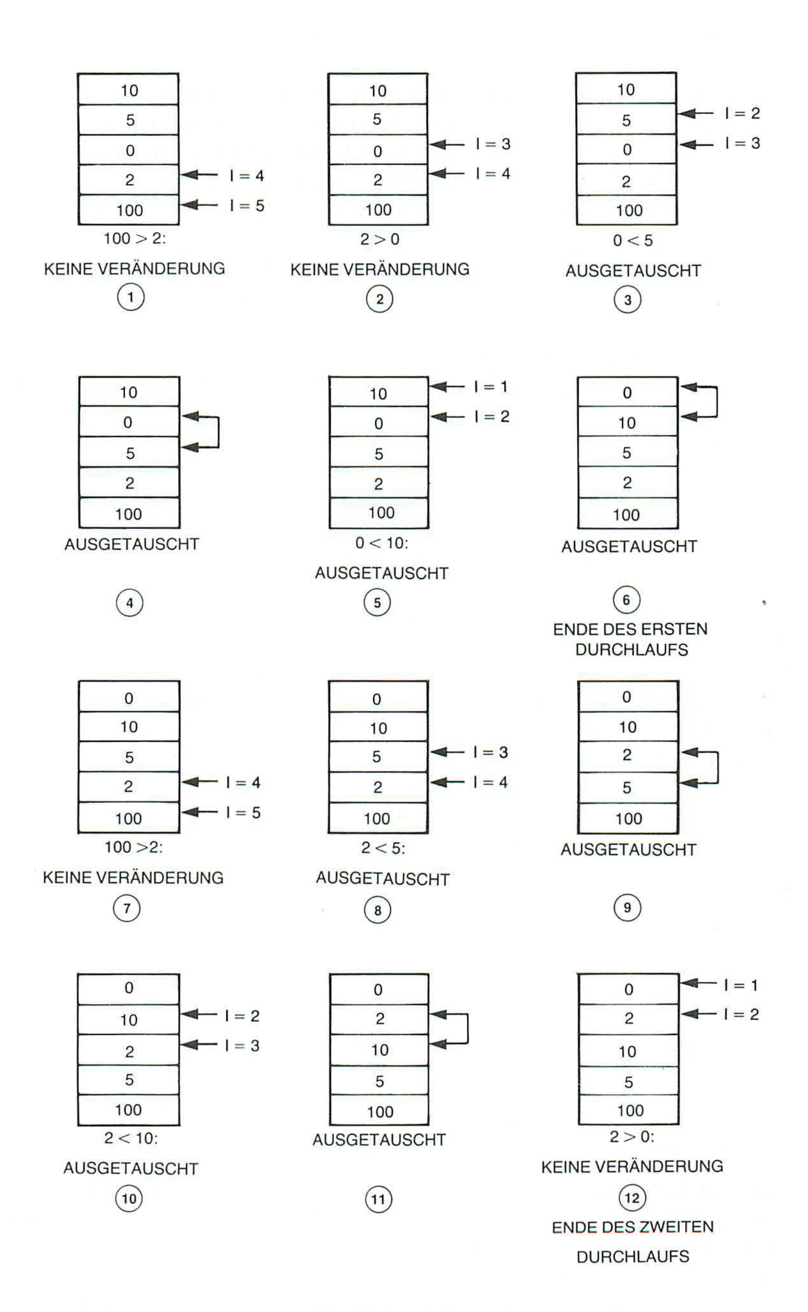


Bild 9-46: Ein Beispiel zum „Bubble-Sort“

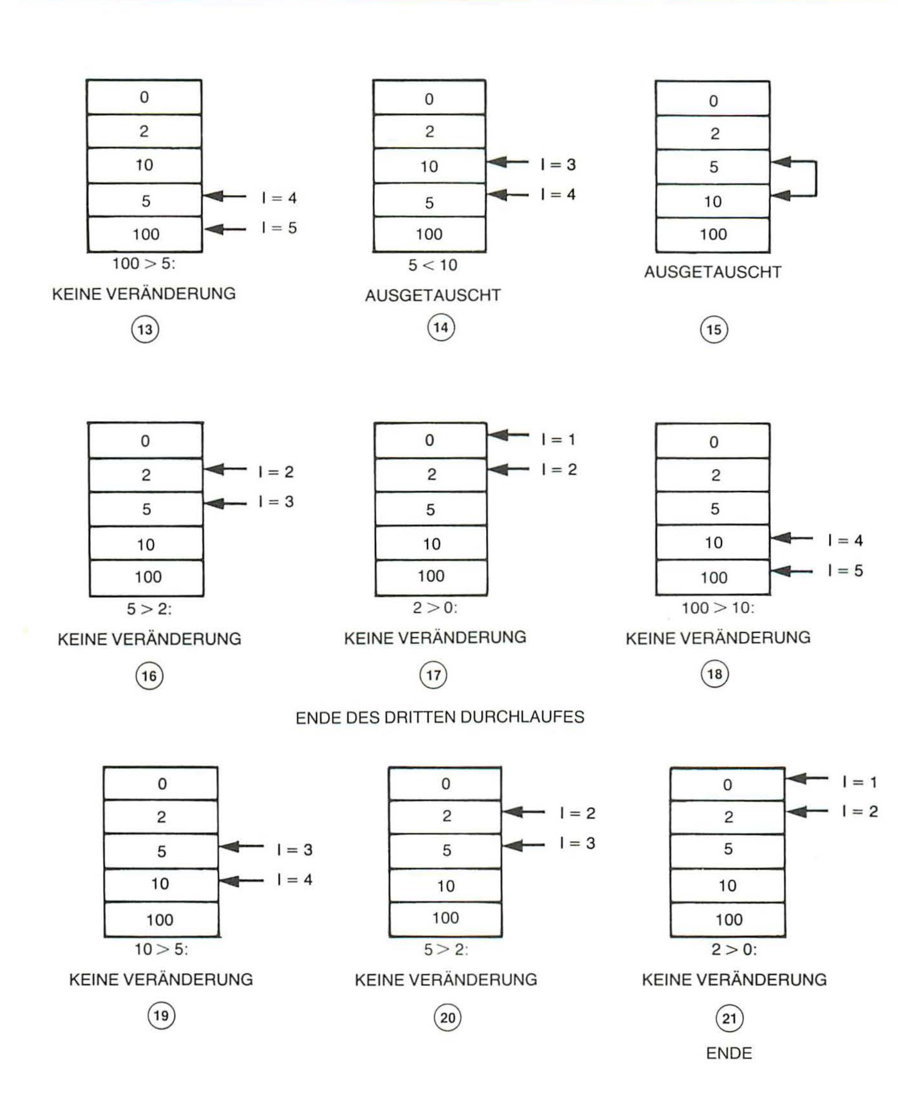


Bild 9-46: Ein Beispiel zum „Bubble-Sort" (Fortsetzung)

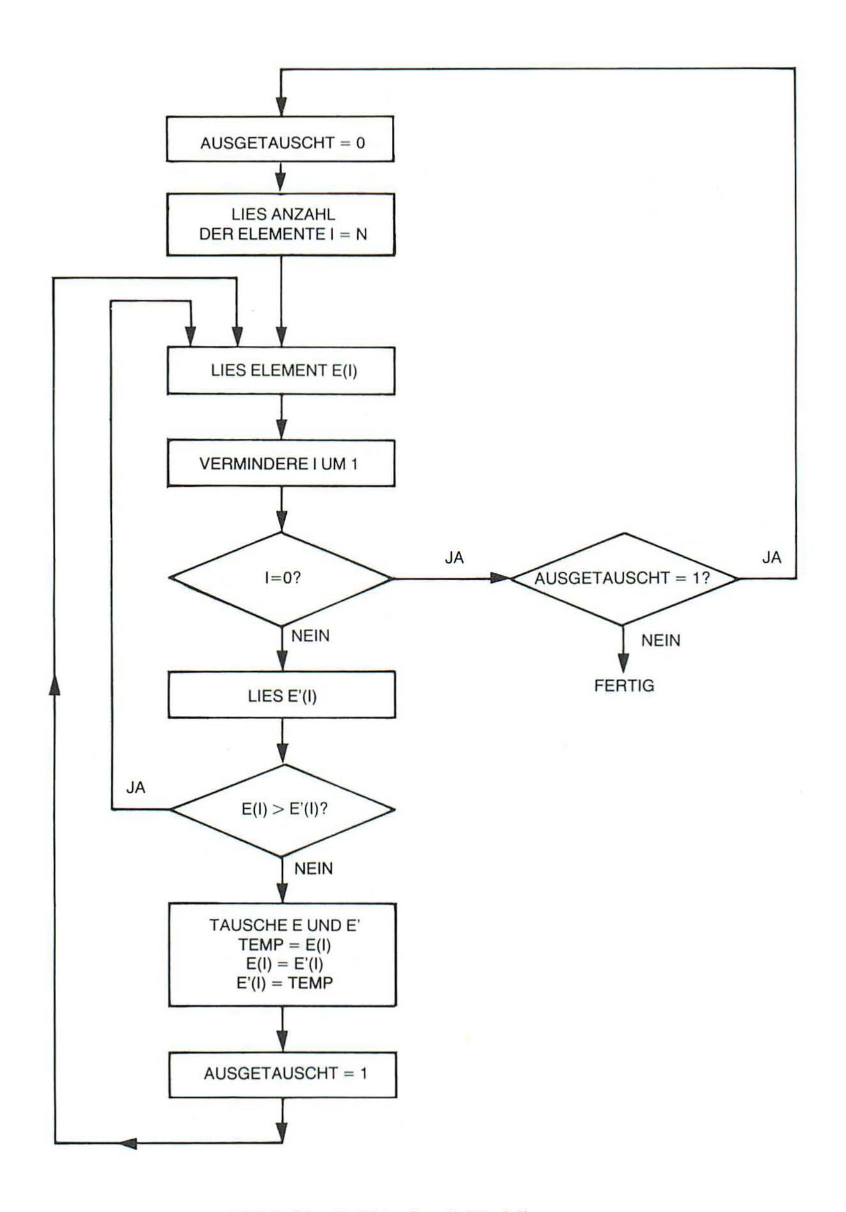


Bild 9-47: „Bubble-Sort": Flußdiagramm

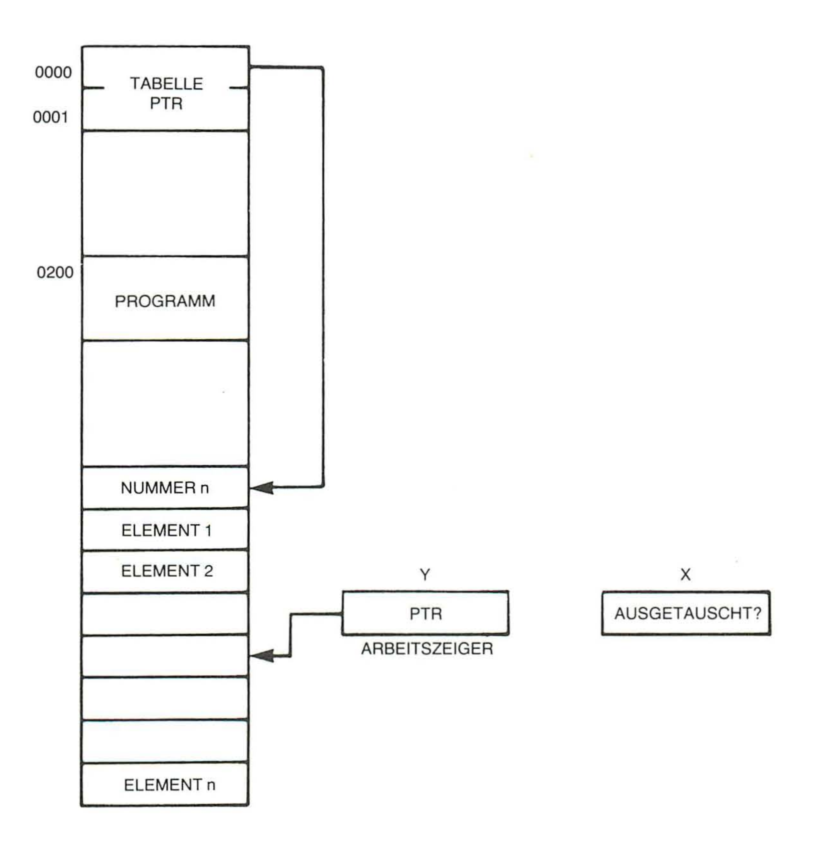


Bild 9-48: „Bubble-Sort“: Speicherbelegung

```
LOC    CODE       LINE

                  ; BUBBLE SORT PROGRAMM
                  ;
0000                     * = $0
                  ;
0000   00 06      TAB    .WORD $600      ; ZEIGER AUF DEN ANFANG DES ZU SORTIERENDEN FELDES.
                  ;
0002                     * = $200
                  ;
0200   A2 00      SORT   LDX #0          ; SETZE 'VERTAUSCHUNG ERFOLGT' AUF 0.
0202   A1 00             LDA (TAB,X)
0204   A8                TAY             ; Y-REGISTER ENTHAELT ANZAHL DER ELEMENTE.
0205   B1 00      LOOP   LDA (TAB),Y     ; LIES ELEMENT E(I).
0207   88                DEY             ; ANZAHL NOCH ZU LESENDER ELEMENTE HERUNTERZAEHLEN.
0208   F0 12             BEQ FINISH      ; KEINE WEITEREN ELEMENTE MEHR ZU BEARBEITEN !
020A   D1 00             CMP (TAB),Y     ; VERGLEICHE GELESENES ELEMENT MIT DEM NAECHSTEN.
020C   B0 F7             BCS LOOP        ; WEITER, FALLS GELESENES ELEMENT GROESSER!
020E   AA         EXCH   TAX             ; FALLS NICHT GROESSER, VERTAUSCHE BEIDE ELEMENTE...
020F   B1 00             LDA (TAB),Y
0211   C8                INY
0212   91 00             STA (TAB),Y
0214   8A                TXA
0215   88                DEY
0216   91 00             STA (TAB),Y
0218   A2 01             LDX #1          ; SETZE 'VERTAUSCHUNG ERFOLGT' AUF 1.
021A   D0 E9             BNE LOOP        ; SCHLEIFE WIEDERHOLEN, FALLS WEITERE ELEMENTE ZU
021C                                     ;  BEARBEITEN.
021C   8A         FINISH TXA             ; 'VERTAUSCHUNG ERFOLGT' ?
021D   D0 E1             BNE SORT        ; JA, WEITER !
021F   60                RTS             ; NEIN, FERTIG.
0220                     .END
```

Bild 9-49: „Bubble-Sort": Programm

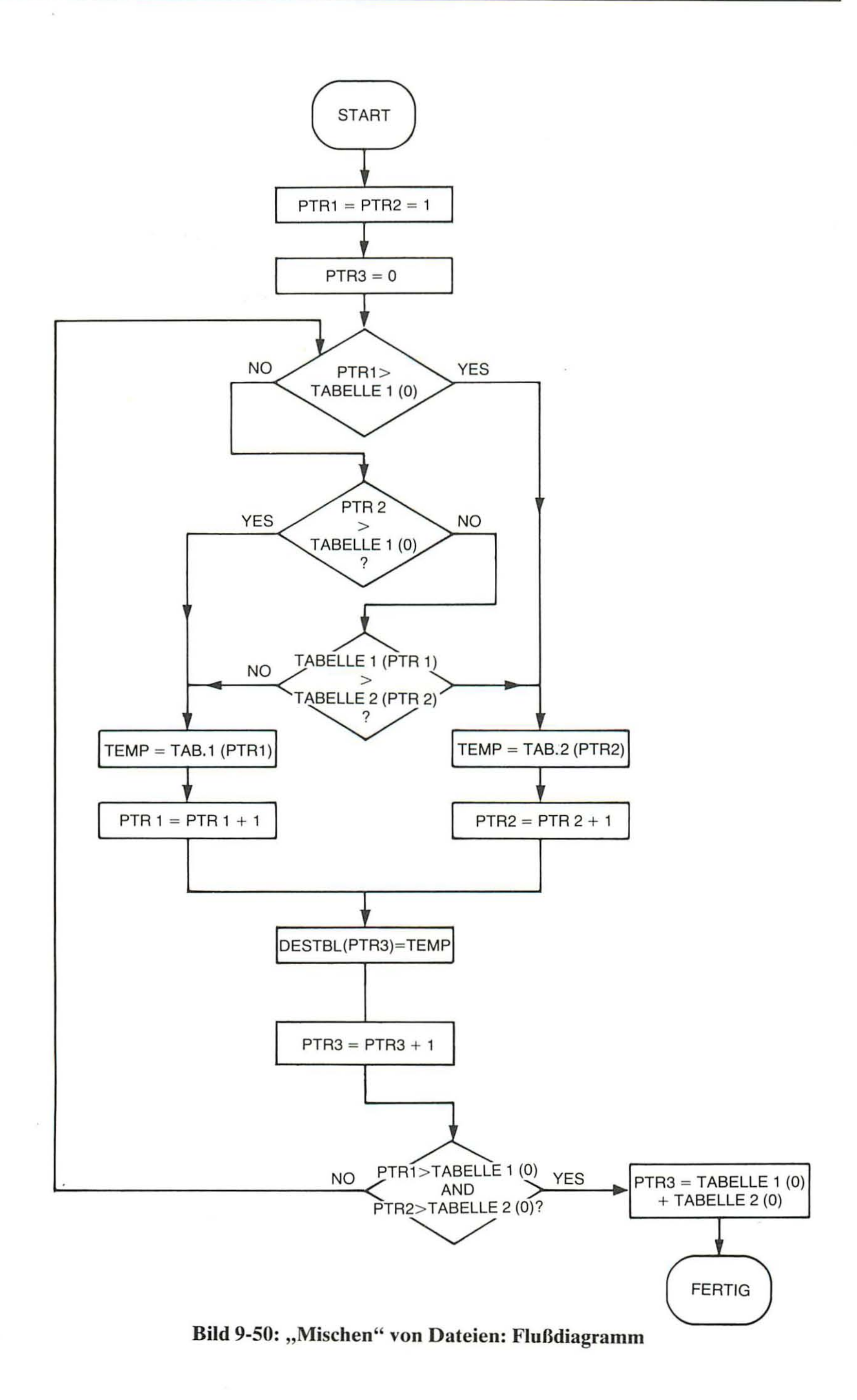


Bild 9-50: „Mischen“ von Dateien: Flußdiagramm

Ein Merge-Algorithmus

Ein weiteres verbreitetes Problem ist das Verschmelzen (merging) zweier Datensätze zu einem dritten. Wir wollen hier annehmen, wir hätten zwei geordnete Tabellen, die wir zu einer Gesamttabelle zusammenfassen wollen. Die Länge jeder der beiden Tabellen ist kleiner als 256 Bytes (je eine Speicherseite). In beiden Tabellen gibt das erste Byte die Zahl der Einträge an.

Der Algorithmus zum Verschmelzen zweier Tabellen ist in Bild 9-50 wiedergegeben. Die zugehörige Speicherorganisation zeigt Bild 9-51 und Bild 9-52 das Programm. Vor dem Aufruf müssen die Ausgangsadressen TABLE1 und TABLE2 sowie die Zieladresse DESTBL (destination table – Zieltabelle) angegeben sein.

Der Algorithmus selbst ist unkompliziert. Es gibt zwei Arbeitszeiger PTR1 und PTR2, die auf Quellentabellen zeigen. PTR3 bezeichnet die sich daraus ergebende Tabelle.

Es wird jeweils ein Paar aus einem Eintrag von TABLE1 und einem von TABLE2 verglichen. Der kleinere von beiden wird in DESTBL übertragen und der zugehörige Arbeitszeiger inkrementiert. Der Prozeß wird so lange fortgesetzt, bis PTR1 und PTR2 am jeweiligen Tabellenende angekommen sind.

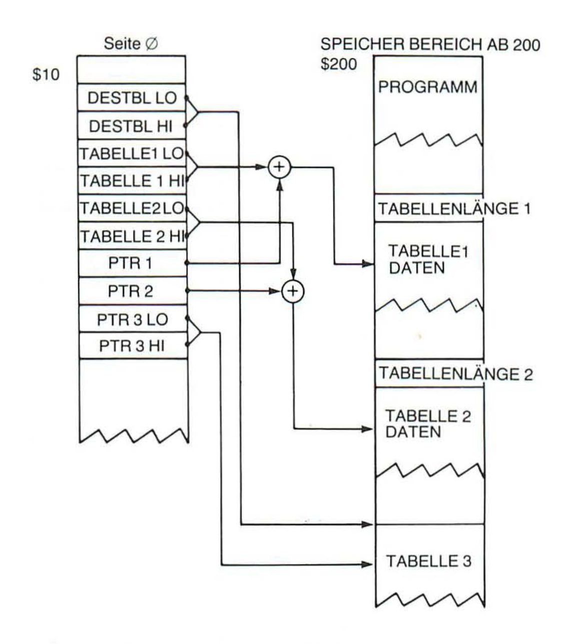


Bild 9-51: „Merging" von Dateien: Speicherbelegung

```
LOC     CODE         LINE

                     ; VERSCHMELZEN ZWEIER DATENSAETZE.
                     ; ZWEI VORSORTIERTE TABELLEN WERDEN DURCH VERSCHMELZUNG
                     ; SORTIERT UND IN EINER DRITTEN TABELLE ABGELEGT.
                     ; JEDE DER BEIDEN QUELLTABELLEN KANN BIS ZU 256 BYTES,
                     ; D.H. EINE SPEICHERSEITE, UMFASSEN.
                     ; DAS JEWEILS ERSTE ELEMENT DER BEIDEN QUELLTABELLEN
                     ; MUSS DEREN LAENGE ENTHALTEN.
                     ; DIE LAENGE DER ZIELTABELLE LIEGT BEIM RUECKSPRUNG IN 'PTR3'.
                     ;
0000                         * = $10
0010    00 00        DESTBL  * = *+ 2         ; ZEIGER AUF ANFANG DER ZIELTABELLE.
0012    00 00        TABLE1  * = *+ 2         ; ZEIGER AUF QUELLTABELLE 1.
0014    00 00        TABLE2  * = *+ 2         ; ZEIGER AUF QUELLTABELLE 2.
0016    00           PTR1    * = *+ 1         ; INDEX FUER TABLE1.
0017    00           PTR2    * = *+ 1         ; INDEX FUER TABLE2.
0018    00 00        PTR3    * = *+ 2         ; INDEX FUER DESTBL.
                     ;
001A                         * = $200
                     ;
0200    A5 11                LDA DESTBL+1     ; PTR3 AUF ZIELTABELLE SETZEN...
0202    85 19                STA PTR3+1
0204    A5 10                LDA DESTBL
0206    85 18                STA PTR3
0208    A9 01                LDA #1           ; INDIZES AUF ANFANG DER JEW. QUELLTABELLE
020A    85 16                STA PTR1         ;  SETZEN, DABEI LAENGENANGABE UEBERSPRINGEN...
020C    85 17                STA PTR2
020E    A2 00                LDX #0           ; INDEXREGISTER LOESCHEN.
0210    A1 14        COMPR   LDA (TABLE2,X)   ; LAENGE TABLE2 <
0212    C5 17                CMP PTR2         ;  INDEX ?
0214    90 19                BCC TKTB1        ; FALLS JA, ARBEITE MIT TABLE1 WEITER.
0216    A1 12                LDA (TABLE1,X)   ; LAENGE TABLE1 <
0218    C5 16                CMP PTR1         ;  INDEX ?
021A    90 0A                BCC TKTB2        ; FALLS JA, ARBEITE MIT TABLE2 WEITER.
021C    A4 16                LDY PTR1         ; HOLE NAECHSTES ELEMENT
021E    B1 12                LDA (TABLE1),Y   ;  AUS TABLE1,
0220    A4 17                LDY PTR2         ; UND VERGLEICHE ES MIT
0222    D1 14                CMP (TABLE2),Y   ;  DEM NAECHSTEN AUS TABLE2.
0224    90 09                BCC TKTB1        ; WAEHLE ERSTERES AUS, FALLS ES KLEINER IST.
0226    A4 17        TKTB2   LDY PTR2         ; WAEHLE ELEMENT AUS TABLE2...
0228    B1 14                LDA (TABLE2),Y
022A    E6 17                INC PTR2         ; SETZE INDEX WEITER.
022C    4C 35 02             JMP STORE        ; SPEICHERE DAS ELEMENT AB.
022F    A4 16        TKTB1   LDY PTR1         ; WAEHLE ELEMENT AUS TABLE1...
0231    B1 12                LDA (TABLE1),Y
0233    E6 16                INC PTR1         ; INDEX WEITERSETZEN.
0235    81 18        STORE   STA (PTR3,X)     ; LEGE AUSGEWAEHLTES ELEMENT IN ZIELTABELLE AB.
0237    E6 18                INC PTR3         ; INDEX WEITERSETZEN...
0239    D0 02                BNE CC
023B    E6 19                INC PTR3+1
023D    A1 12        CC      LDA (TABLE1,X)   ; NOCH WEITERE ELEMENTE IN TABLE1 VORHANDEN ?
023F    C5 16                CMP PTR1
0241    B0 CD                BCS COMPR        ; JA, WEITER.
0243    A1 14                LDA (TABLE2,X)   ; NEIN! NOCH WEITERE ELEMENTE IN TABLE2 VORHANDEN ?
0245    C5 17                CMP PTR2
0247    B0 C7                BCS COMPR        ; JA, WEITER.
0249    A9 00                LDA #0           ; NEIN! VERSCHMELZUNG BEENDET.
024B    85 19                STA PTR3+1       ; BERECHNE TABELLENLAENGE IN ZIELTABELLE DURCH
024D    18                   CLC              ;
024E    A1 12                LDA (TABLE1,X)   ;  ADDITION DER LAENGEN DER QUELLTABELLEN...
0250    61 14                ADC (TABLE2,X)
```

```
LOC     CODE         LINE

0252    85 18                STA PTR3
0254    90 04                BCC CCC
0256    A9 01                LDA #1
0258    85 19                STA PTR3+1
025A    60           CCC     RTS              ; FERTIG !
025B                         .END
```

Bild 9-52: „Merging“ von Dateien: Programm

Zusammenfassung

Es wurden die Grundkonzepte allgemeiner Datenstrukturen als auch praxisorientierter Anwendungsbeispiele vorgestellt.
Der 6502 fördert mit seinen leistungsfähigen Adressierungsmöglichkeiten die Handhabung komplexer Datenstrukturen. Die Kompaktheit der besprochenen Programme beweist das.
Außerdem wurden besondere Algorithmen zur Hashing-Technik, zur Sortierung und zur Verschmelzung (merging) besprochen, die zur Lösung komplexer Probleme notwendig werden können.
Der Programmieranfänger braucht sich mit den Einzelheiten von Datenstrukturen nicht unnötig zu belasten. Ein sinnvolles Programmieren von nichttrivialen Algorithmen setzt jedoch ein fundiertes Verständnis dieser Strukturen und Techniken voraus.
Die in diesem Kapitel vorgestellten Beispiele mögen dem Leser eine Hilfestellung zum Verstehen und Lösen der bei den üblichen Datenstrukturen auftauchenden Probleme bieten.

KAPITEL 10
PROGRAMMENTWICKLUNG

Einleitung

Die meisten Programme, die wir bis jetzt untersucht haben, sind von Hand ohne jegliche Soft- oder Hardwarehilfen erstellt worden (ausgenommen die Ausdrucke im letzten Kapitel). Die einzige Verbesserung gegenüber direkt binärer Kodierung bestand darin, daß wir mnemonische Symbole, die Darstellungsform der Assemblersprache, verwendet haben. Zum komfortableren und flexibleren Erstellen von Software ist ein fundiertes Verständnis der Programmentwicklungshilfen in Hard- und Software unabdingbar. Ziel dieses Kapitels ist es, diese Hilfen vorzustellen und ihre Funktion zu erläutern.

Programmieralternativen

Es gibt drei grundlegende Alternativen zum Erstellen von Programmen: binäre und hexadezimale Kodierung, Kodierung in Assemblersprache und Kodierung in höheren Programmiersprachen. Sehen wir uns diese Möglichkeiten näher an.

1. Hexadezimale Kodierung

Das Programm wird normalerweise in Assemblersprache geschrieben. Die meisten preisgünstigen Einkartencomputer verfügen jedoch nicht über einen Assembler. Ein solcher ist ein Programm, das automatisch die symbolische Programmdarstellung in die zur Abarbeitung erforderliche binäre Form bringt. Wenn kein Assembler zur Verfügung steht, muß man diese Übersetzung von Hand vornehmen. Dabei ist die binäre Darstellungsform anstrengend zu schreiben und zu lesen und somit sehr fehlerintensiv. Man benutzt deshalb bei der Übersetzung hexadezimale Kodierung. In Kapitel 1 haben wir gesehen, daß eine Hexadezimalziffer gerade vier Bits zusammenfaßt. Damit genügen zur Wiedergabe eines Bytes zwei Hexadezimalziffern. So ist beispielsweise die 6502-Befehlstabelle im Anhang in hexadezimaler Form geschrieben. Kurz gesagt: Wenn die verfügbaren finanziellen Mittel beschränkt sind und kein Assembler zur Verfügung steht, dann muß man das Programm eben von Hand in hexadezimale Form übersetzen. Das ist jedoch nur bei kurzen Programmen vertretbar (sagen wir bei einem Umfang von 10 bis 100 Befehlen). Bei längeren Programmen ist dieses Vorgehen zeitraubend und fehlerintensiv, so daß es kaum einsetzbar wird. Fast alle Einkartencomputer verlangen jedoch die Programmeingabe in hexadezimaler Form. Sie verfügen über keinen Assembler und besitzen aus Kostengründen keine Volltastatur.

Hexadezimale Kodierung ist kein sehr angenehmer Weg, ein Programm in den Rechner zu bekommen. Sie ist ausschließlich billig. Man hat hier die Kosten für einen Assembler und eine Volltastatur gegen die Zusammenarbeit bei der Programmeingabe in den Speicher eingetauscht. Das ändert jedoch nichts an der Art, in der das Programm selbst erstellt wird. Man schreibt es auch hier am besten in Assemblersprache, wodurch es ohne großen Aufwand untersucht und änderbar wird.

2. Programmieren auf Assemblerebene

Programmieren auf Assemblerebene schließt Programme ein, die sowohl in hexadezimaler als auch in symbolischer Assemblersprache in den Speicher eingegeben werden sollen. Hexadezimale Eingabe entspricht dem im vorigen Kapitel Gesagten; wir wollen uns daher gleich auf die Programmeingabe in Assemblersprache konzentrieren. Man benötigt für diesen Zweck ein Assemblerprogramm im Rechner. Dieses liest jeden mnemonischen Befehl und übersetzt ihn in das zugehörige Ein-, Zwei- oder Dreibytemuster. Ein guter Assembler bietet darüber hinaus eine ganze Reihe weiterer Möglichkeiten zur Programmerstellung. Diese sollen weiter unten genau betrachtet werden. Insbesondere gibt es die Möglichkeit mit Hilfe von sog. „Pseudobefehlen" die Arbeit des Assemblers unmittelbar zu beeinflussen. Dazu gehört z.B. die Vorgabe eines bestimmten Werts an ein Symbol. Man kann symbolische Adressierung benutzen und so zu einer durch ein Label gegebenen Programmstelle verzweigen. Das macht es überflüssig, während der Fehlerbeseitigungsphase jedesmal, wenn ein Befehl herauszunehmen oder einzufügen war, das ganze Programm zwischen Sprungbefehl und Sprungziel neu zu schreiben. Der Assembler weist den Labels bei der Übersetzung automatisch die richtige Adresse zu. Darüber hinaus gestattet ein Assembler dem Benutzer, das Programm in symbolischer Form von Fehlern zu be-

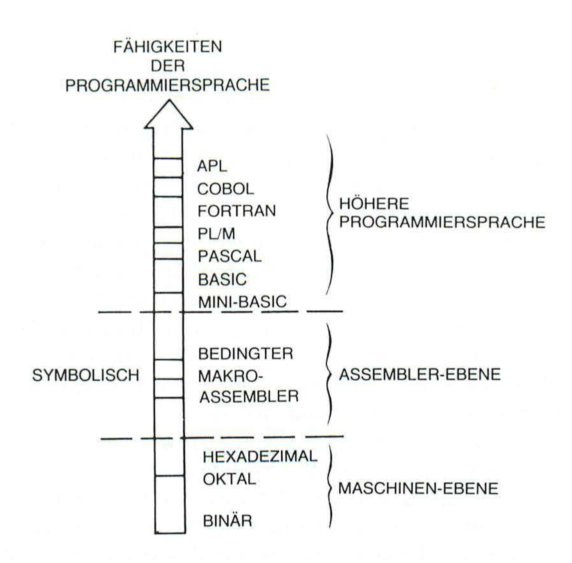


Bild 10-1: Ebenen von Programmiersprachen

freien. Man kann dazu noch einen *Disassembler* benutzen, der den Speicherinhalt aus der binären Form in die symbolischen Befehle der Assemblerebene zurückübersetzt. Wir werden unten die verschiedenen Softwarehilfen, die ein System normalerweise zur Programmerstellung bietet, genauer betrachten. Sehen wir uns vorher aber noch eine dritte Programmalternative an.

3. Höhere Programmiersprachen

Man kann ein Problem auch in einer höheren Programmiersprache abfassen, wie z.B. in BASIC, APL, Pascal und anderen. Die zur Programmierung in solche Sprachen nötigen Techniken werden von speziellen Büchern behandelt und sollten hier nicht weiter betrachtet werden. Wir wollen diese Möglichkeit hier nur ganz kurz ansprechen. Eine höhere Programmiersprache bietet sehr leistungsfähige und bequem anzuwendende Befehle an, mit denen das Programmieren sehr viel einfacher und schneller wird. Diese Befehle müssen von komplexen Programmen in die letztlich vom Computer gebrauchten binären Maschinenbefehle übersetzt werden. In der Regel wird jeder der höheren Programmierbefehle in eine große Zahl von binären Maschinenbefehlen übersetzt. Man unterscheidet bei den Programmen, die dies vollziehen, zwischen *Compilern* und *Interpretern.* Ein Compiler übersetzt alle Befehle aus der höheren Programmiersprache am Stück in den Objektkode (d.h. die ausführbaren Maschinenbefehle), der dann unabhängig vom Übersetzungsprogramm abgearbeitet werden kann. Ein Interpreter dagegen sieht sich jeden Befehl an und bringt den Computer dazu, ihn auszuführen, bevor er zum nächsten Befehl übergeht. Ein Interpreter hat den Vorteil, daß Fehler sofort sichtbar und behebbar sind. Er arbeitet „interaktiv" mit dem Benutzer. Doch arbeitet er sehr viel langsamer als ein Compiler. Wir wollen hier jedoch auf dieses Thema nicht näher eingehen und uns lieber wieder mit dem Programmieren auf Maschinen- und Assemblerebene beschäftigen.

Softwarehilfen

Sehen wir uns die wesentlichsten Softwaremöglichkeiten an, die in einem vollständigen System zur Erleichterung der Programmentwicklung vorhanden sind (oder zumindest sein sollten). Einige dieser Programme wurden bereits vorgestellt und sollen unten nur noch zusammengefaßt werden.

Ein *Assembler* ist ein Programm, das die mnemonische Befehlswiedergabe in ihre binären Entsprechungen übersetzt. Normalerweise wird jeder symbolische Befehl in einen binären Befehl (zu 1, 2 oder 3 Bytes) übersetzt. Der sich daraus ergebende binäre Programmkode wird *Objektkode* genannt (im Unterschied zum *Quellenkode,* aus dem er erstellt worden ist). Er kann vom Mikrocomputer unmittelbar abgearbeitet werden. Zusätzlich zu dieser Arbeit erzeugt der Assembler einen vollständigen Programmausdruck (listing), in dem die Befehle sowohl symbolisch als auch hexadezimal zusammen mit den belegten Speicheradressen angegeben sind (vgl. die Beispiele im vorigen Kapitel). Des weiteren liefert ein guter Assembler noch die Zuordnungstabellen für die verschiedenen freien Labels und schließlich ein geordnetes Symbolverzeichnis mit den Adressen, an denen diese Symbole im Programm auftauchen.

Ein *Compiler* übersetzt eine höhere Programmiersprache in binären Objektkode, der vom Computer abgearbeitet werden kann.

Ein *Interpreter* hat ähnliche Aufgaben. Er übersetzt ebenfalls Befehle einer höheren Sprache, führt sie aber sofort aus, anstatt den erzeugten Kode für spätere Abarbeitung zu speichern. In aller Regel erzeugt er sogar keinen Objektkode, sondern arbeitet die den höheren Befehlen zugeordneten Routinen unmittelbar ab.

Ein *Monitor* ist ein zum Ausnutzen der Hardwaremöglichkeiten des Geräts unabdingbares Programm. Er überwacht (monitors) die Eingabegeräte ständig und verwaltet des weiteren die übrigen Einheiten. So muß z.B. ein Minimalmonitor, wie man ihn auf einem Einkartenmikrocomputer mit Hexadezimaltastatur und LED-Anzeige findet, ständig die Tastatur auf eine Eingabe hin abfragen und die angeforderten Inhalte auf der LED-Anzeige wiedergeben. Außerdem muß er eine Reihe einfacher Befehle von der Tastatur her erkennen können, wie START, STOP, CONTINUE (Weitermachen), LOAD MEMORY (Speicherstelle laden) und EXAMINE MEMORY (Speicherstelle überprüfen). Auf einem Großrechner wird der Monitor oft als Organisations- oder Ablaufprogramm (executive program) bezeichnet. Wenn zusätzlich eine komplexe Dateiverwaltung oder Aufgabenorganisation durchgeführt wird, nennt man den Gesamtumfang der hierfür nötigen Routinen ein *Betriebssystem* (operating system). Wenn die bearbeitete Datei auf Disketten steht, heißt das zur Verwaltung nötige Programm auch *Diskettenbetriebssystem* DOS – disk operating system).

Ein *Editor* ist ein zur Eingabe und Bearbeitung von Texten aller Art (insbesondere auch für Programmtexte) entwickeltes Programm. Es gestattet bequem Zeichen einzugeben, an den vorhandenen Text anzufügen oder in ihn einzufügen, erlaubt Zeilen einzufügen und zu löschen und kann nach Einzelzeichen oder Zeichenketten suchen. Derartige „Editiermöglichkeiten" sind eine wichtige Unterstützung zur einfachen und leistungsfähigen Texteingabe.

Ein *Debugger* ist eine für die Fehlersuche *(debugging)* in Programmen gedachte Hilfe. In der Regel ist es so, daß bei einem nicht arbeitenden Programm zunächst keinerlei Hinweis vorhanden ist, wo die Ursache dafür liegen könnte. Man setzt in so einem Fall Haltepunkte (breakpoints) in das Programm ein, die die Abarbeitung an einem bestimmten Punkt anhalten und dem Programmierer die Möglichkeit geben, die Register- und Speicherinhalte zu untersuchen. Dies ist die Grundaufgabe eines Debuggers. Er gestattet das Programm zu unterbrechen oder seine Arbeit wieder aufnehmen zu lassen und ermöglicht die Untersuchung und Änderung von Register- und Speicherinhalten. Ein guter Debugger bietet noch weitere Möglichkeiten, wie z.B. die Speicherinhalte nach Wahl in symbolischer, hexadezimaler oder binärer Form auszugeben und gestattet die Dateneingabe in allen diesen Formen.

Ein *Lader* oder *bindender Lader* (linking loader) dient dazu, Objektkodes ganz oder in Teilen an bestimmte Plätze im Speicher zu laden und die inneren Beziehungen im geladenen Programm, die durch symbolische Zeiger angegeben werden, dem Speicherbereich entsprechend zu justieren, so daß die verschiedenen Programmteile miteinander zusammenarbeiten können. Man benutzt ihn, um Programme oder Blöcke in beliebigen Speicherbereichen einsetzen zu können.

Ein *Simulator* dient dazu, die Arbeitsweise eines Geräts, normalerweise des Mikroprozessors, zu Testzwecken bei der Programmentwicklung nachzubilden. Man kann auf diese Weise das Programm jederzeit unterbrechen, es modifizieren und für die Dauer des Tests im RAM-Bereich halten (auch wenn es im Originaltext in den ROM-Bereich gehört). Die Nachteile eines Simulators sind:

1. Normalerweise wird nur der Prozessor nachgebildet, nicht die E/A-Einheiten.
2. Die Abarbeitung erfolgt langsam, man muß in simulierter Zeit arbeiten. Das macht es unmöglich, Echtzeitsysteme auszutesten, da selbst wenn die Programmlogik in Ordnung ist, hier noch Synchronisationsprobleme auftreten können.

Ein *Emulator* ist ein Simulator, der in Echtheit arbeiten kann. Es handelt sich hier um eine Hardwarenachbildung, bei der der Prozessor des Testgeräts durch einen besonders aufgebauten Prozessor ersetzt wird. Dadurch kann man alle Einzelheiten in der Arbeit des Geräts verfolgen.

Hilfsroutinen sind im Grunde alle die Routinen, die man gerne vom Hersteller geliefert bekommen hätte. Das kann Multiplikation und Division umfassen, sowie andere arithmetische Operationen, Blockverschiebungsprogramme, Zeichentestroutinen, Ein/Ausgabesteuerungen („Treiber") und mehr.

Der Programmiervorgang

Damit können wir uns den typischen Ablauf der Erstellung eines Programms auf Assemblerebene ansehen. Wir nehmen dazu an, daß alle üblichen Software-Entwicklungshilfen vorhanden sind. Wenn ein System nicht alle davon bieten sollte, ist es nach wie vor möglich, Programme zu entwickeln, es geht nur nicht mehr so bequem, und der Zeitaufwand, bis das Programm läuft, wird sich u.U. beträchtlich vergrößern.

Normalerweise entwickelt man erst einen Algorithmus und legt die zu verwendenden Datenstrukturen für das zu lösende Problem fest. Darauf entwirft man einen umfassenden Satz von Flußdiagrammen (oder äquivalenter Beschreibungen), die die Programmlogik wiedergeben. Schließlich werden diese Diagramme in die Assemblersprache des Mikroprozessors übertragen, was die eigentliche Kodierphase darstellt.

Dann muß man das Programm in den Computer eingeben. Wir wollen uns dazu die hierbei benutzbaren Hardwarealternativen ansehen.

Das Quellenprogramm wird mit Hilfe eines Editors in den RAM-Bereich des Systems geschrieben. Sowie ein zusammenhängender Programmabschnitt eingegeben ist, sollte er getestet werden.

Zunächst wird der Assembler eingesetzt. Ist er nicht bereits im System fest eingebaut (im ROM-Bereich), so muß er vorher von außen (etwa von einer Diskette) geladen werden. Er assembliert das Programm, d.h. übersetzt es in den binären Objektkode, der in der Folge abgearbeitet werden kann.

In aller Regel arbeitet kein Programm auf Anhieb richtig. So setzt man zunächst an strategisch wichtigen Stellen, an denen man die Zwischenergebnisse einfach auf Richtigkeit überprüfen kann, Haltepunkte (breakpoints) ein. Für diesen Zweck wird der Debugger verwendet. Man gibt die gewünschten Haltepunkte vor und startet dann das Programm mit einem passenden Debugger-Befehl (z.B. mit „Go"). Das Programm stoppt automatisch an den vorbestimmten Punkten. Dort kann man dann mit Hilfe des Debuggers die richtige Arbeit durch Untersuchen der Register- und Speicherinhalte überprüfen. Ist die Sache in Ordnung, kann man bis zum nächsten Haltepunkt weitergehen. Andernfalls muß im gerade abgearbeiteten Text irgendwo ein Fehler stecken. Normalerweise nimmt man sich hier den Programmausdruck vor, den der Assembler erstellt hat und sieht nach, ob die Assemblerbefehle richtig formuliert sind. Ist hier nichts zu finden, so liegt ein logischer Fehler vor, der zum Flußdiagramm

zurückverweist. Wir wollen allerdings annehmen, daß die Arbeit der Flußdiagramme von Hand ausgetestet wurde und nach Lage der Dinge in Ordnung sein müßte. So liegt der Fehler höchstwahrscheinlich in falschen Kodeeingaben. Auf alle Fälle muß man einen Teil des Programms abändern. Wenn das Quellenprogramm noch im Speicher steht, so können wir einfach den Editor wieder aufrufen, mit ihm die betreffenden Programmzeilen ändern und dann die eben beschriebenen Schritte erneut durchgehen. In einigen Systemen oder bei sehr langen Programmabschnitten reicht der Platz im Speicher für das Quellenprogramm und den Objektkode zusammen nicht aus, so daß man es vor der Übersetzung erst in einen externen Speicher (üblicherweise eine Kassette oder eine Diskette) retten muß. In diesem Fall muß der Objektkode (und oft auch der Editor) bei einem Fehler erst wieder von außen geladen werden, bevor man die notwendigen Änderungen durchführen kann.

Dieses Verfahren wird so lange wiederholt, bis das Programm die richtigen Ergebnisse erarbeitet. Bedenken Sie aber unbedingt, daß auch hier Vorbeugen besser als Heilen ist. Ein sauberer Entwurf führt üblicherweise sehr schnell zu einem arbeitenden Programm (man hat im wesentlichen nur Tippfehler oder offensichtliche Kodierfehler zu korrigieren). Ein auf die Schnelle erstelltes Programm aber braucht eine unverhältnismäßig lange Zeit zum Austesten (manche derartige „Programme" arbeiten nie!). Kurz gesagt, der anfängliche Mehraufwand für einen sorgfältigen Programmentwurf zahlt sich beim Austesten doppelt und dreifach aus.

Dieses Vorgehen erlaubt das Austesten der Programmgrundfunktionen, gestattet jedoch normalerweise keine Aussage darüber, ob das Programm auch unter Echtzeitbedingungen und mit den vorgesehenen Ein/Ausgabegeräten zusammen arbeitet. Zum Austesten dieser Zusammenarbeit kann man den direkten Weg einschlagen, das logisch fertiggetestete Programm in ein EPROM laden, dieses im vorgesehenen Computer installieren und dann nachsehen, ob es funktioniert oder nicht.

Es gibt hier einen besseren Weg, den Einsatz eines sogenannten *In-Circuit-Emulators* (ein Emulator in der Anwenderschaltung). Ein solcher benutzt einen 6502- oder sonstigen Prozessor, um den 6502 der Anwenderschaltung in (nahezu) Echtheit nachzubilden. Der Emulator enthält ein Verbindungskabel mit einem 40-poligen Stecker, der einfach anstelle des 6502 in die Anwenderschaltung gesteckt wird. Er erzeugt genau dieselben Signale wie der 6502 – höchstens ein wenig langsamer. Der Hauptvorteil ist, daß das Programm jetzt im RAM-Bereich des Entwicklungssystems, das den Emulator enthält, stehen kann. Das gestattet auch in dieser E/A-Testphase den Einsatz aller Möglichkeiten des Entwicklungssystems (Editor, Debugger, Symbolmöglichkeiten, Dateisystem).

Ein guter Emulator bietet noch weitere Möglichkeiten, unter denen die „Trace"-Möglichkeit hervorgehoben werden muß. Ein Trace (wörtlich: Spur) ist die Aufzeichnung der zuletzt ausgeführten Befehle und/oder der letzten Zustände der verschiedenen Systembusse vor einem Haltepunkt. So gestattet ein Trace, sich wie in einem Film die Ereignisse anzuschauen, die vor dem Haltepunkt eingetreten sind. Ja, man kann in ausgebauten Systemen sogar ein Oszilloskop beim Eintreten einer bestimmten Adreß- oder Bikombination triggern. Diese Möglichkeit der „Rückblende" ist außerordentlich wertvoll, denn in aller Regel ist es beim Auftreten des Fehlers bereits zu spät. Der Befehl oder die Daten, die den Fehler auslösten, liegen vor der Entdeckung des Fehlers selbst. Mit einem Trace kann man den Programmteil auffinden, der den Fehler verursacht hat. Und wenn die Rückblendespanne nicht ausreichen sollte, so setzt man halt einfach den Haltepunkt etwas früher an.

Damit sind wir mit unserer Beschreibung der normalen Schritte bei der Programmentwicklung fertig. Sehen wir uns nun an, was es an Hardwarealternativen für die Programmerstellung gibt.

ROM	RAM
INITIALER LADER (BOOTSTRAP)	ASSEMBLER ODER ÜBERSETZER ODER INTERPRETIERER
TASTATUR-TREIBER	DOS
ANZEIGE-TREIBER	EDITOR ODER FEHLERSUCH-PROGRAMM ODER SIMULATOR
TELETYPE-TREIBER	SYSTEM-ARBEITSSPEICHER (UND STAPEL)
KASSETTEN-TREIBER	BENUTZERPRGRAMM
KOMMANDO-INTERPRETIERER	BENUTZER-ARBEITSSPEICHER
HILFSROUTINEN	
ELEMENTARES FEHLERSUCH-PROGRAMM	
ELEMENTARER EDITOR	

Bild 10-2: Eine typische Speicheraufteilung

Die Hardwarealternativen

1. Einkartenmikrocomputer

Der Einkartenmikrocomputer bietet Programmiermöglichkeiten zu geringsten Kosten an. Normalerweise verfügt er über eine Hexadezimaltastatur, einige Funktionstasten und sechs LEDs, mit denen man Adressen und Daten anzeigen kann. Da er nur wenig Speicherplatz bietet, ist normalerweise kein Assembler vorhanden. Bestenfalls besitzt er einen kleinen Monitor und – außer ein paar grundlegenden Befehlen – keine Edier- und Fehlersuch-(debugging)-möglichkeiten. Alle Programme müssen in hexadezimaler Form eingegeben werden und werden in hexadezimaler Form angezeigt. Theoretisch hat ein Einkartencomputer dieselbe Leistungsfähigkeit wie andere Computer auch. Wegen seines eingeschränkten Speicherangebots und seiner Tastatur bietet er nicht alle Möglichkeiten größerer Systeme und verlängert so die Programmentwicklung beträchtlich. Der Umstand, den die Entwicklung hexadezimaler Programme mit sich bringt, macht einen Einkartencomputer für Lehr- und Trainingszwecke

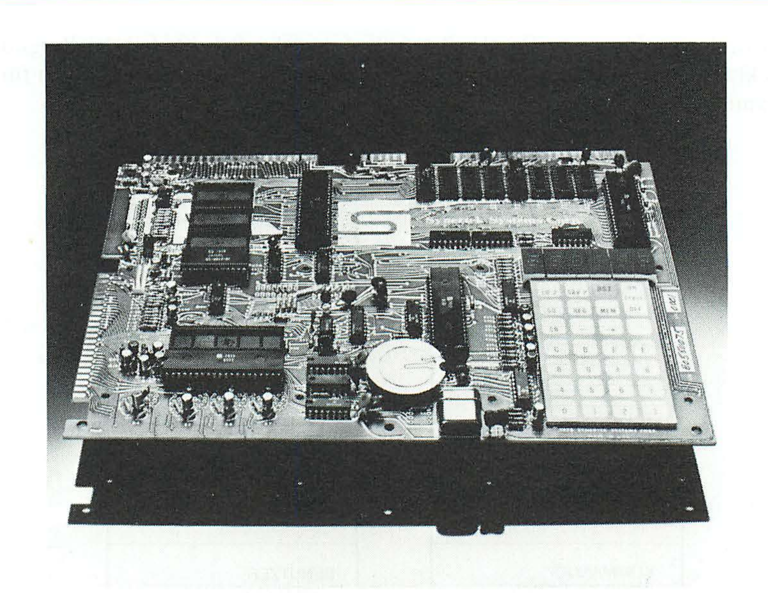


Bild 10-3: Der SYM1 ist ein typischer Einkartencomputer

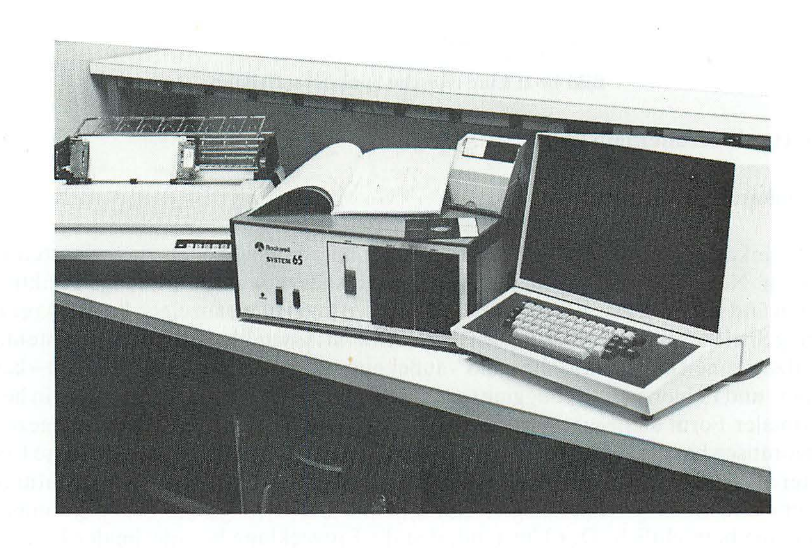


Bild 10-4: Das „System 65“ von Rockwell ist ein Entwicklungssystem

geeignet, wo die Programme nur beschränkte Länge haben sollen. Einkartencomputer sind vermutlich der billigste Weg, das Programmieren durch die eigene Praxis zu lernen. Sie können jedoch zur Entwicklung komplexerer Programme kaum sinnvoll eingesetzt werden, es sei denn, man erweitert sie mit zusätzlichen Speicherkarten und den üblichen Softwarehilfen.

2. Das Entwicklungssystem

Ein Entwicklungssystem ist ein Mikrocomputersystem, ausgestattet mit einem beträchtlichen RAM-Bereich (32 x 48 K) und allen benötigten Ein/Ausgabeeinheiten wie einem Bildschirmgerät, einem Drucker, Diskettenstationen, üblicherweise auch einem ROM-Programmierer und vielleicht auch einem In-Circuit-Emulator. Ein Entwicklungssystem ist speziell auf die Programmentwicklung im industriellen Bereich ausgerichtet. Es bietet üblicherweise alle, oder doch die meisten, der im vorigen Abschnitt beschriebenen Softwarehilfen an. Im Prinzip ist es das ideale Instrument zur Softwareentwicklung.
Bis vor kurzem war ein Mikrocomputer-Entwicklungssystem dahingehend beschränkt, daß keine höheren Programmiersprachen, die Compiler- oder Interpreterprogramme erfordern, bei der Programmentwicklung eingesetzt werden konnten, da diese sehr viel Speicherplatz belegen. Dies änderte sich jedoch in der letzten Zeit mit der allgemeineren Verfügbarkeit von BASIC, Pascal und anderen höheren Sprachen (insbesondere PL/65 von Rockwell) auch für 6502-Systeme.
Der Schwerpunkt der Leistungsfähigkeit von Entwicklungssystemen liegt jedoch auf Assemblerebene. Hier bietet jedes von ihnen alle notwendigen Eigenschaften an. Jedoch werden gute Entwicklungssysteme nur in relativ kleinen Stückzahlen verkauft und sind daher recht teuer.

3. Hobbycomputer

Die Hardware eines Hobbycomputers entspricht der eines Entwicklungssystemes. Der Hauptunterschied liegt darin, daß der durchschnittliche Hobbycomputer nicht über die ausgefeilten Softwareentwicklungshilfen verfügt, die ein industrielles Entwicklungssystem bietet. So beinhalten die meisten Hobbycomputer beispielsweise nur elementare Assembler, minimale Editoren, rudimentäre Dateiverwaltungsmöglichkeiten, keine Möglichkeit PROMs zu programmieren, keinen In-Circuit-Emulator, keinen leistungsstarken Debugger. Sie stellen einen Zwischenschritt zwischen Einkartencomputer und professionellen Entwicklungssystemen dar. Jedoch sind sie bei der Entwicklung von Programmen mittlerer Komplexität wahrscheinlich der beste Kompromiß, da sie den Vorzug relativ geringer Kosten mit einem brauchbaren Angebot von Programmentwicklungswerkzeugen verbinden, selbst wenn diese nicht so leistungsfähig wie die großer Entwicklungssysteme sind.
Hier ist für die nähere Zukunft jedoch einige Veränderung zu erwarten. So nähern sich die ausgebauten Hobbycomputer in ihren Möglichkeiten immer mehr den professionellen Entwicklungssystemen an. Der Unterschied zwischen einem durchschnittlichen Programmentwicklungssystem und einem leistungsfähigen Hobbycomputer verschwindet immer mehr. Diese Entwicklung ist bei anderen Prozessortypen, bei denen früher eine Softwarestandardisierung eingesetzt hat, ausgeprägter als zur Zeit noch beim 6502, zeichnet sich aber auch hier deutlich ab.

4. Timesharingsysteme

Verschiedene Firmen vermieten Sichtgeräte, die an sogenannte Timesharing-Netzwerke angeschlossen werden können. Diese Terminals teilen sich in die Arbeit eines Großcomputers und bieten so all die Möglichkeiten großer Systeme. Für nahezu alle Mikroprozessortypen gibt es sogenannte *Cross-Assembler* auf Timesharingsystemen. Ein Cross-Assembler ist nichts weiter als ein Assembler, der beispielsweise auf einer IBM 370 Kode für den 6502 erzeugen kann. Formal ist ein Cross-Assembler ein Assembler für einen Mikroprozessor X, der auf einem Prozessor Y abgearbeitet wird. Die Art des verwendeten Computers ist dabei unwichtig. Der Benutzer schreibt ein Programm in 6502-Assemblersprache, und der Cross-Assembler übersetzt es in einen 6502-Objektkode. Die einzige Schwierigkeit liegt darin, daß das Programm nicht unmittelbar abgearbeitet werden kann. Es kann zwar – so vorhanden – von einem simulierten Prozessor auf dem Großcomputer abgearbeitet werden, kann aber so keine Ein/Ausgabeeinheiten bedienen. Wegen dieses Nachteils und wegen der relativ hohen Kosten ist Timesharing nur im industriellen Bereich praktikabel.

5. Hauseigene Großcomputer

Wenn ein großer Computer im Haus verfügbar ist, kann man oft auch dort Cross-Assembler zur Programmentwicklung einsetzen. Wenn dieser Computer Timesharing-Eigenschaften besitzt, so entspricht diese Möglichkeit genau dem oben Dargestellten. Wenn nur sogenannter Stapelbetrieb (batch processing) möglich ist, bei dem die Programme im Computerzentrum abgegeben und später dort abgeholt werden müssen, ist dies wahrscheinlich die unbequemste Methode der Programmentwicklung, da dieses Vorgehen sehr viel Zeit kostet.

Schalterkonsole oder nicht?

Eine Schalterkonsole ist ein oft zur Fehlersuche in Programmen herangezogenes Werkzeug. Traditionell dient es zur bequemen Anzeige von Register- und Speicherinhalten. Die meisten dieser Funktionen lassen sich mit entsprechenden Hilfsprogrammen auch über die Bildschirmstation erledigen, die genausogut die Registerinhalte binär abbilden kann. Zusätzlich hat man hier aber den Vorteil, daß man (wenn die Softwaremöglichkeiten vorhanden sind) hier auch beliebig hexadezimal, dezimal oder symbolisch arbeiten kann. Der Hauptnachteil der Entwicklung vom Bildschirmgerät aus ist, daß man statt Bedienen eines Schalters nun für einen Befehl mehrere Tasten drücken muß. Die Kosten für eine Schalterkonsole sind jedoch recht beachtlich, so daß die meisten der neueren Mikrocomputer dieses Werkzeug zugunsten der Bedienung über Bildschirmgeräte und Monitorprogramm weglassen. Der Wert einer Schalterkonsole liegt meist nur in der eigenen Vorstellung, die auf vergangener Erfahrungen und der verbreiteten menschlichen Trägheit beruht. Sie ist jedenfalls in den allermeisten Fällen entbehrlich.

Zusammenfassung der Hardwaremöglichkeiten

Man kann drei Fälle unterscheiden. Wenn Sie nur über ein eingeschränktes Budget verfügen und doch das Programmieren lernen möchten, so sollten Sie einen Einkar-

tencomputer kaufen. Mit seiner Hilfe können Sie alle einfachen Programme aus diesem Buch erstellen und noch eine ganze Menge mehr. Sie werden jedoch bald auf die Grenzen eines solchen Systems stoßen, wenn Sie Programme mit mehr als ein paar hundert Befehlen erstellen wollen.

Wenn Sie den Mikrocomputer in der Industrie einsetzen wollen, dann brauchen Sie ein voll ausgebautes Entwicklungssystem. Jede Lösung darunter verursacht ein beträchtliches Ansteigen der Entwicklungszeit und damit zusätzliche Kosten. Man muß sich hier eindeutig entscheiden: Investitionen in die Hardware oder Investitionen in die Entwicklungsdauer mit allen begleitenden Kosten. Sind die zu entwickelnden Programme recht einfach, so braucht man natürlich weniger in die Hardware investieren. Sind die Programme jedoch voraussichtlich relativ komplex, so kann man kaum irgendwelche Hardwareeinsparungen beim Kauf eines Entwicklungssystems rechtfertigen. Die sich ergebenden Mehrkosten bei der Programmentwicklung übertreffen die Einsparung bei weitem.

Für den privaten Gebrauch dürfte ein Hobbycomputer in den meisten Fällen ausreichende, wenn auch eingeschränkte Möglichkeiten bieten. Doch zeichnet sich allmählich auch gute Entwicklungssoftware für Hobbycomputer ab. Die meisten – insbesondere der „Fertigsysteme“ – haben jedoch immer noch eingeschränkte Möglichkeiten und werden sie voraussichtlich auf einige Zeit hinaus behalten.

Untersuchen wir nun das Programm im Detail, auf das man am wenigsten verzichten kann: den Assembler.

Der Assembler

Wir haben im ganzen Buch bis jetzt die Assemblersprache des 6502 benutzt, ohne jedoch die formalen Regeln und Definitionen dafür anzugeben. Das soll nun hier nachgeholt werden. Ein Assembler ist so entwickelt, daß der Benutzer sich einer bequemen symbolischen Programmdarstellung bedienen kann, wobei zugleich diese mnemonischen Befehle auf einfache Art und Weise in ihre binäre Entsprechungen umgewandelt werden können.

Format eines Assemblerprogramms

Wenn wir ein Quellenprogramm für den Assembler erstellen, so müssen wir zwischen verschiedenen Spalten unterscheiden (Bild 10-5). Sie enthalten folgende Informationen:

Label: Kann frei bleiben. Hier steht die symbolische Adresse des auf der Zeile folgenden Befehls.

Befehl (instruction): Umfaßt den Befehlskode (opcode) und den evtl. nötigen Operanden. (Man kann das Operandenfeld auch getrennt davon behandeln.)

Kommentar: Dieses ganz rechts stehende Feld ist beliebig und dient zu Erläuterungen des Programms.

Wenn dem Assembler der Quellenkode übergeben ist, erzeugt er ein sogenanntes *Listing,* einen Ausdruck des vollständigen Programms. Dazu benutzt der Assembler drei weitere Spalten, üblicherweise links von der Quellenprogrammauflistung. Bild 10-6 zeigt ein Beispiel dazu. Ganz links ist die Zeilennummer angegeben. Hier werden alle im Quellenprogramm auftretenden Zeilen durchnumeriert. (Das ist nicht

wesentlich für das Programm selbst. Es gibt Assembler, die keine Zeilennummer bieten. Es gestattet aber den leichteren Zugriff auf bestimmte Programmzeilen beim Einsatz des – üblicherweise zeilenorientierten – Editors.)

	HEX BEFEHL			SYMBOLISCHE(R)			KOMMENTARE
ADR.	1	2	3	MARKE	OPCODE	OPERAND	

Bild 10-5: Ein Formular zur Mikroprozessorprogrammierung

Die Spalte rechts daneben gibt die Adresse an. Diesen Wert hat der Programmzähler, wenn er den in der Zeile stehenden Befehl aus dem Speicher übernimmt.
Es folgt die hexadezimale Darstellung des in der Zeile angegebenen Befehls. Die in dieser Spalte stehenden Werte sind der eigentliche Inhalt des Programmspeichers nach der Übersetzung des Quellenkodes. Sie geben den abzuarbeitenden Objektkode wieder.
Das ist eine der Einsatzmöglichkeiten eines Assemblers. Selbst wenn wir ein Programm auf einem Einkartencomputer austesten wollen, der nur hexadezimale Bedienung kennt, können wir das Programm in Assemblersprache abfassen, Haben wir dann Zugriff auf den Assembler in einem anderen System, so können wir das Programm von diesem übersetzen lassen und den hexadezimalen Objektkode im Programmausdruck anschließend in den Textcomputer eintippen. Dieses einfache Beispiel zeigt den Wert zusätzlicher Softwaremöglichkeiten.

Tabellen

Wenn ein Assembler das Quellen- in ein Objektprogramm übersetzt, führt er zwei Hauptaufgaben durch:
1. Er übersetzt die mnemonischen Befehle in ihre binären Entsprechungen.
2. Er übersetzt die symbolischen Angaben für Konstanten und Adressen in ihre binären Werte.

Um die Fehlersuche zu erleichtern, fassen die meisten Assembler am Ende des Programms die verwendeten Symbole und ihre binären Werte in einer alphabetisch geordneten Liste zusammen, der *Symboltabelle*.
Einige Versionen geben als zusätzliche Arbeitserleichterung nicht nur die Symbole und ihre Adresse an, sondern auch noch die Zeilennummer, in denen die Symbole auftauchen.

Fehlermeldungen

Der Assembler findet bei seiner Arbeit die im Quellenprogramm aufgetretenen *Syntaxfehler* auf und gibt sie im Programmausdruck oder gesondert an. Typische derartige Fehler sind: undefinierte Symbole, doppelt definierte Labels, ungültige Befehlskodes, unzulässige Adressen, unzulässige Adressierungsart verwendet. Natürlich wären noch viel mehr derartige Fehlermeldungen erwünscht und werden oft auch angegeben. Das wechselt jedoch von Assembler zu Assembler.

Die Assemblersprache

Die Befehlskodes (opcodes) sind bereits definiert worden. Wir wollen hier noch die Symbole, Konstanten und Operatoren beschreiben, die als Teil der Assemblersyntax im Programm auftauchen können.

Symbole

Symbole werden zur Wiedergabe numerischer Werte (Daten oder Adressen) verwendet. Üblicherweise können diese Symbole 6 alphanumerische Zeichen umfassen, von denen das erste ein Buchstabe sein muß. Es gibt noch eine weitere Einschränkung: Die 56 vom 6502 benutzten Befehlskodes und die Registernamen (A, X, Y, S, P) dürfen in der Regel nicht als Symbole benutzt werden.

Wert eines Symbols

Labels sind spezielle Symbole, denen vom Programmierer kein ausdrücklicher Wert zugewiesen werden braucht (und in der Regel auch nicht darf). Sie erhalten beim Assemblieren automatisch den Wert der Adresse, die im Programmausdruck in der zugehörigen Befehlszeile steht. Alle anderen als Konstanten oder Adreßangaben benutzten Symbole müssen jedoch vom Programmierer vor ihrem Einsatz definiert werden. Man benutzt in der Syntax der meisten 6502-Assembler zu diesem Zweck das Gleichheitszeichen, das somit als *Pseudobefehl* dient. (Ein Pseudobefehl taucht in der Spalte der Befehlskodes auf, wird aber nicht unmittelbar in einen binären Programmbefehl übersetzt. Es ist eine Anweisung [directive] für den Assembler selbst.)
So läßt sich zum Beispiel mit der folgenden Zeile der Konstanten ALPHA der hexadezimale Wert „A000“ zuweisen:

```
ALPHA    =    $A000
```

Die übrigen Pseudobefehle werden später besprochen.

```
LINE #   LOC     CODE    LINE

0057     0342  A9 00             LDA  #$00
0058     0344  8D 0B A0          STA  ACR1      ;TURN BOTH TIMERS OFF
0059     0347  8D 0B AC          STA  ACR2
0060     034A  A2 20             LDX  #OFFDEL   ;GET TONES-OFF DELAY CONSTANT
0061     034C  20 55 03  OFF     JRS  DELAY     ;DELAY WHILE TONE IS OFF
0062     034F  CA                DEX
0063     0350  D0 FA             BNE  OFF
0064     0352  4C 02 03          JMP  DIGIT     ;GO BACK FOR NEXT DIGIT OF PHONE NUMBER
0065     0355            ;
0066     0355            ;THIS IS A SIMPLE DELAY ROUTINE FOR THE TONE ON AND OFF PERI
0067     0355            ;
0068     0355  A9 FF     DELAY   LDA  #DELCON   ;GET DELAY CONSTANT
0069     0357  38        WAIT    SEC            ;DELAY FOR THAT LONG
0070     0358  E9 01             SBC  #$01
0071     035A  D0 FB             BNE  WAIT
0072     035C  60                RTS
0073     035D            ;
0074     035D            ;THIS IS A TABLE OF THE CONSTANTS FOR THE TONE FREQUENCIES
0075     035D            ;FOR EACH TELEPHONE DIGIT.  THE CONSTANTS ARE TWO BYTES
0076     035D            ;LONG, LOW BYTE FIRST.
0077     035D            ;
0078     035D  13        TABLE  .BYTE $13,$02,$76,$01 ;TWO TONES FOR '0'
0078     035E  02
0078     035F  76
0078     0360  01
0079     0361  CD               .BYTE $CD,$02,$9E,$01 ;TWO TONES FOR '1'
0079     0362  02
0079     0363  9E
0079     0364  01
0080     0365  CD               .BYTE $CD,$02,$76,$01 ;       '2'
0080     0366  02
0080     0367  76
0080     0368  01
0081     0369  CD               .BYTE $CD,$02,$53,$01 ;             '3'
0081     036A  02
0081     036B  53
0081     036C  01
0082     036D  89               .BYTE $89,$02,$9E,$01 ;        '4'
0082     036E  02
0082     036E  9E
0082     0370  01
0083     0371  89               .BYTE $89,$02,$76,$01          '5'
0083     0372  02
0083     0373  76
0083     0374  01
0084     0375  89               .BYTE $89,$02,$53,$01 ;       '6'
0084     0376  02
0084     0377  53
0084     0378  01
0085     0379  4B               .BYTE $4B,$02,$9E,$01 ;       '7'
0085     037A  02
0085     037B  9E
0085     037C  01
0086     037D  4B               .BYTE $4B,$02,$76,$01 ;    '8'
0086     037E  02

LINE #   LOC     CODE    LINE

0086     037E  76
0086     0380  01
0087     0381  4B               .BYTE $4B,$02,$53,$01 ;          '9'
0087     0382  02
0087     0383  53
0087     0384  01
0088     0385                   .END

SYMBOL   TABLE

SYMBOL     VALUE

ACRI       A00B   ACR2   AC0B   DELAY   0353   DELCON  00FF
DIGIT      0302   NOEND  030A   NUMPTR  0000   OFF     034C
OFFDEL     0020   ON     033C   ONDEL   0040   PHONE   0300
T1CH       A005   T1LH   A007   T1LL    A004   T2CH    AC05
T2LH       AC07   T2LL   AC04   TABLE   035D   WAIT    0357

END OF ASSEMBLY
```

Bild 10-6: Beispiel eines Assemblerausdrucks (Assemblerlisting)

Konstanten

Konstanten werden üblicherweise in dezimaler, hexadezimaler oder binärer Form wiedergegeben. Außer bei Dezimalzahlen muß der Wertangabe ein Zeichen vorangehen, das die verwendete Basis angibt. Um den dezimalen Wert 18 in den Akkumulator zu laden, genügt es

LDA # 18 (#gibt an, daß es sich um eine Konstante handelt)

zu schreiben.

Eine Hexadezimale Zahl erhält den Vorsatz „$“.
Eine Oktalzahl erhält den Vorsatz „@“.
Eine Dualzahl erhält den Vorsatz „%“.

Um den binären Wert „11111111“ in den Akkumulator zu laden, können wir z.B. schreiben:

LDA # %1111111

Man kann außerdem ASCII-Zeichen als Konstante benutzen. Ältere Assembler forderten, die ASCII-Angabe in einfache Anführungsstriche (Apostrophe) einzuschließen. Neuere Versionen verringern die Tipparbeit etwas, indem entsprechend den anderen Definitionen nur gefordert wird, daß der Wertangabe ein einfaches Apostroph vorangeht. Um z.B. den Buchstaben „S“ in seiner ASCII-Kodierung in den Akkumulator zu laden, schreibt man in der neueren Version:

LDA # 'S

Nach wie vor schließt ein Apostroph allerdings den angegebenen Teil ab. Um ein solches Zeichen selbst als Konstante angeben zu können, hat man die Vereinbarung getroffen, daß es doppelt geschrieben werden muß. So lädt z.B. der folgende Befehl den ASCII-Kode für „'“ in den Akkumulator:

LDA#'''

Übung 10.1:
Laden die Befehle LDA #'5 und LDA # $5 denselben Wert in den Akkumulator?

Operatoren

Um das Erstellen symbolisch geschriebener Programme weiter zu vereinfachen, erlauben die meisten Assembler den Einsatz von Operatoren zur Wertangabe. So sollte mindestens „+“ und „−“ vorhanden sein, daß man z. B. folgendes festlegen kann:

LDA ADR1

und

LDA ADR1 + 1

Man muß dabei vor Augen halten, daß der Ausdruck ADR1 + 1 vom Assembler zur Berechnung der nächsten Speicheradresse herangezogen wird. Die Berechnung erfolgt bei der *Assemblierung* (assembly time) und nicht bei der Programmausführung (program execution time).
Des weiteren sind üblicherweise noch mehr Operatoren verfügbar, wie Multiplikation und Division, mit denen die Adressierung von Tabellen vereinfacht wird. Außerdem gibt es noch spezialisierte Operatoren, wie „>“ und „<“, die von einem 16-Bit-Wert die höherwertige (>) bzw. die niederwertige Hälfte (<) auswählen.
Normalerweise werden diese Zahlen positiv aufgefaßt. Negative Zahlen müssen ausdrücklich in ihrer hexadezimalen Entsprechung angegeben werden.

Schließlich gibt es bei fast allen 6502-Assemblern mit dem Stern (★) ein spezielles Zeichen, das den Wert der Adresse in der gegebenen Zeile wiedergibt. Man liest es als „gerade vorliegende Stelle“ (current location), womit es den an dieser Programmstelle erreichten Programmzählerstand bezeichnet.

Übung 10.2:
Was ist der Unterschied zwischen den folgenden Befehlen?

```
LDA  %10101010
LDA  #%10101010
```

Übung 10.3:
Was bewirkt der folgende Befehl?

```
BMI  ★-2
```

Pseudobefehle

Pseudobefehle (assembler directives) sind besondere Anweisungen im Programmtext, die die Arbeitsweise des Assemblers steuern sollen. Einige dieser Anweisungen weisen bestimmten Symbolen oder Speicherplätzen einen festen Wert zu. Andere steuern den Gang der Assemblierung oder geben die Form des Programmausdrucks vor.
Betrachten wir als Beispiel die neun Pseudobefehle, die der Assembler des Entwicklungssystems von Rockwell („System 65“) bietet. Sie lauten:
=,.BYT, .WOR, .GBY, .PAGE, .SKIP, .OPT, .FILE und .END.

Die Zuordnungsanweisung
Man benutzt das Gleichheitszeichen dazu, einem Symbol einen Wert zuzuweisen, beispielsweise so:

```
BASIS   =   $1111
★       =   $1234
```

Der erste Pseudobefehl weist BASIS den hexadezimalen Wert 1111 zu. Jedesmal, wenn im Programm danach BASIS verwendet wird, ersetzt der Assembler es durch hexadezimal 1111.
Der zweite Befehl setzt die Adresse der betrachteten Zeile auf hexadezimal 1234. Das bewirkt, daß die folgenden Befehle ab Speicherstelle 1234 abgelegt werden.

Übung 10.4:
Schreiben Sie einen Pseudobefehl, aufgrund dessen das danach folgende Programm mit Adresse 0 beginnt.

Initialisieren von Speicherplatz
Diesem Zweck dienen drei Pseudobefehle: .BYT, .WOR und .GBY.
Der erste, .BYT, weist den ab der gegebenen Adresse kommenden Speicherstellen die als Argument angegebenen Werte byteweise zu. Mit

```
RESERV    .BYT 'SYBEX'
```

werden z.B. die ASCII-Kodes für die Buchstaben S, Y, B, E und X nacheinander im Speicher abgelegt.
16-Bit-Adressen werden mit .WOR im Speicher abgelegt, wobei das niederwertige Byte zuerst kommt. So bewirkt z.B.

```
ZIELE     .WOR $1234, $2345
```

daß man ab Speicheradresse ZIELE folgende Bytefolge findet:
34, 12, 45 und 23.
Der Pseudobefehl .GBY entspricht .WOR mit der Ausnahme, daß der 16-Bit-Wert jetzt mit dem höherwertigen Byte zuerst gespeichert wird. Man benutzt ihn besser für 16-Bit-Daten anstatt für 16-Bit-Adressen.
Die nächsten drei Pseudobefehle steuern den Programmausdruck.

Programmausdruck

Hierzu stehen die Befehle .PAGE, .SKIP und .OPT bereit.
Durch .PAGE schließt der Assembler die gerade bearbeitete Ausdruckseite ab und beginnt eine neue. Man kann im .PAGE-Befehl zusätzlich einen Seitentitel vorgeben, der automatisch bis zur nächsten Titelvorgabe am Kopf jeder Seite angegeben wird.
Der Pseudobefehl .SKIP bewirkt eine oder mehrere Leerzeilen im Programmausdruck. Wie .PAGE läßt er sich zur Gliederung des Druckbilds und damit zur Verbesserung der Übersichtlichkeit einsetzen. Man kann im Argument angeben, wieviele Leerzeilen eingeschoben werden sollen, für 3 Leerzeilen z. B. „.SKIP 3“.
Mit .OPT läßt sich eine von drei Ausgabemöglichkeiten (options) wählen: LIST, GENERATE, ERROR. LIST bewirkt den Ausdruck der gesamten Programmliste. GENERATE druckt den vollständigen hexadezimalen Kode bei .BYT-Befehlen aus (ohne GENERATE werden nur die beiden ersten Bytes angegeben). ERROR bewirkt den alleinigen Ausdruck fehlerhafter Programmteile. SYMBOL gibt an, daß die vollständige Symboltabelle auszudrucken ist. Die gewünschten Möglichkeiten werden als Argument zu .OPT angegeben und werden durch Kommas voneinander getrennt.
Die beiden letzten Pseudobefehle steuern die Eingabe des Quellenfiles.

Eingabesteuerung

Beim Entwickeln eines großen Programms werden die verschiedenen Programmteile üblicherweise getrennt geschrieben und zum Laufen gebracht. Irgendwann wird es aber notwendig, die einzelnen Teile als Gesamtes zu assemblieren. Dazu muß der Assembler nacheinander die Einzelabschnitte aus der Quellenprogrammdatei aufrufen können. Das erste derartige „File" enthält dann den Pseudobefehl .FILE NAME/1, wobei 1 die Nummer der Diskettenstation und NAME der Name der nächsten Quellenaufzeichnung (des nächsten „Quellenfiles") ist. Das nächste File kann dann seinerseits eine Verbindung (link) zu einem folgenden File tragen, bis schließlich am Ende des letzten Files ein Abschlußbefehl stehen muß: .END NAME/1. Hier bezieht sich NAME auf das erste bearbeitete Quellenfile, da der Assembler zwei Durchgänge durch den Quellentext machen muß.

Kommentare

Man kann in den Ausdruck zusätzliche Kommentarzeilen aufnehmen, wenn ihnen ein Strichpunkt (;) vorausgeht. Diese allgemeine Kommentiermöglichkeit ist eine wichtige Sache für eine gute Programmdokumentation. Man kann hier zusammenfassende Erklärungen zu jeder Routine geben, insbesondere zur Register- und Speicherbelegung bei Aufruf und Rücksprung, evtl. Hinweise zur Stapelbelegung, eine allgemeine Funktionsbeschreibung u. ä. Sorgfältige Kommentierung in diesen Zusatzzeilen macht das Gesamtprogramm wesentlich besser versteh- und wartbar.

Befehlsmakros

Die meisten derzeit verfügbaren 6502-Assembler bieten keine Makromöglichkeiten an. Wir wollen hier in der Hoffnung, daß Makrofähigkeiten bald die Regel bei 6502-Assemblern werden, trotzdem erklären, was ein Befehlsmakro ist und worin seine Vorzüge liegen.

Ein Befehlsmakro ist einfach nur eine unter einem gemeinsamen Namen zusammengefaßte Gruppe von Befehlen. Es stellt in der Hauptsache eine Programmiererleichterung dar. Wenn zum Beispiel eine Gruppe aus fünf Befehlen häufig im Programm vorkommt, so kann man für sie ein Befehlsmakro definieren und diesen Namen überall da im Programm verwenden, wo diese fünf Befehle hingehören. Wir können z.B. schreiben:

```
RETTEN    MACRO
                    PHA
                    TXA
                    PHA
                    TYA
                    PHA
          ENDM
```

Jedesmal, wenn wir im Programm dann den „Befehl" RETTEN angeben, setzt der Assembler die fünf Befehle PHA, TXA, PHA, TYA und PHA im Programm ein.

(ENDM bezeichnet das Ende des Befehlsmakros in der Vereinbarung.) Dadurch spart man sich erstens Schreibarbeit und erreicht zum zweiten (bei geschickter Wahl der Makronamen) eine bessere Programmlesbarkeit.
Ein mit Makromöglichkeiten ausgestatteter Assembler heißt Makroassembler.

Befehlsmakro oder Unterprogramm?

Es mag hier so aussehen, als ob ein Befehlsmakro gerade wie ein Unterprogramm eingesetzt würde. Das ist jedoch nicht der Fall. Beim Erzeugen des Objektkodes fügt der Assembler *jedesmal,* wenn er den Befehlsmakronamen entdeckt, die dafür definierte Befehlsfolge in das Programm ein. Es ist gerade so, als wenn man diese Befehlsgruppe jedesmal auch wirklich aufgeschrieben hätte.
Ein Unterprogramm ist etwas davon völlig verschiedenes. Es wird nur *einmal* übersetzt, kann dann aber beliebig oft abgearbeitet werden: Das Programm springt zur Unterprogrammadresse. Es wird keinerlei zusätzlicher Objektkode dafür erzeugt.
Ein Befehlsmakro ist eine Möglichkeit des Assemblers, ein Unterprogramm eine Möglichkeit des Programms. Ein Makro wird bei der Assemblierung abgearbeitet (assemblytime). Ein Unterprogramm bei der Programmausführung (executiontime). Ihre Arbeitsweise ist völlig verschieden.

Makroparameter

Man kann jedem Befehlsmakro eine bestimmte Anzahl von Parametern geben. Betrachten wir dazu folgendes Beispiel:

```
TAUSCH   MACRO  M, N, T
         LDA    M
         STA    T
         LDA    N
         STA    M
         LDA    T
         STA    N
         ENDM
```

Mit diesem Befehlsmakro tauschen die Speicherstellen M und N ihren Inhalt aus, wobei T als Hilfsregister benutzt wird. Da ein solcher Austausch nicht zum Befehlssatz des 6502-Prozessors gehört, kann man ihn durch ein Makro ersetzen. Dabei werden die symbolischen Namen M, N und T in der Makrodefinition beim Makroaufruf durch die entsprechenden aktuellen Speichernamen ersetzt. Nehmen wir an, wir wollten die Speicherstellen ALPHA und BETA austauschen und dazu die Speicherstelle TEMP als Hilfsregister verwenden:

```
TAUSCH ALPHA, BETA, TEMP
```

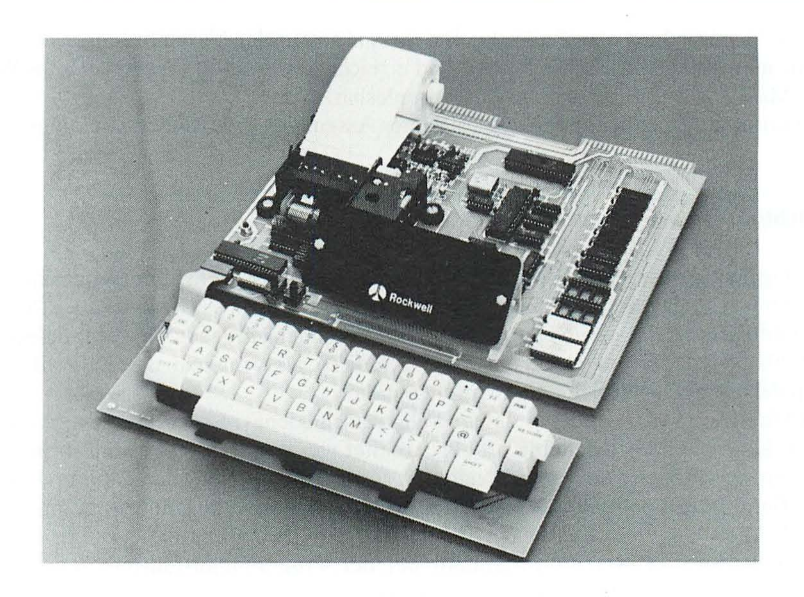


Bild 10-7: Der AIM65 ist ein Einkartencomputer mit Drucker und Volltastatur



Bild 10-8: Ein Hobbycomputer auf 6502-Basis von Ohio Scientific (OSI)

Der Assembler fügt beim Übersetzen folgende Befehlskette in das Programm ein:

```
LDA  ALPHA
STA  TEMP
LDA  BETA
STA  ALPHA
LDA  TEMP
STA  BETA
```

Damit wird der Wert von Befehlsmakros deutlich: Sie sind eine gewaltige Hilfe für den Programmierer, der sich mit ihrer Hilfe neue „Befehle" definieren kann. Damit läßt sich der Befehlssatz des 6502 ganz nach den Anforderungen erweitern. Allerdings muß man dabei berücksichtigen, daß jede Verwendung eines Befehlsmakros das Programm um mehrere Bytes (je nach Zahl der definierten Befehle) verlängert. Ein Befehlsmakro wird deshalb immer langsamer abgearbeitet werden, als ein entsprechender vom Hersteller bereitgestellter Befehl (aber immer noch schneller als ein Unterprogramm!). Wegen ihrer enormen Vereinfachung der Programmierarbeit ist bei längeren Programmen eine Makromöglichkeit höchst wünschenswert.

Weitere Makromöglichkeiten

Man kann diese Grundeigenschaften von Befehlsmakros noch um viele Möglichkeiten erweitern. So könnte man z.B. Makros *verschachteln,* d.h. einen Makroaufruf innerhalb eines anderen Befehlsmakros zulassen. Das würde gestatten, daß sich ein Makro selbst verändert! Beim ersten Aufruf können so z.B. verschiedene Initialisierungsparameter gesetzt werden. Danach ändert das Makro seine Definition und alle folgenden Aufrufe dieses Makronamens führen zu Befehlsketten ohne diese Initialisierungssequenz.

Bedingte Assemblierung

Bedingte Assemblierung ist eine andere Möglichkeit, die man bis jetzt bei den meisten 6502-Assemblern vermißt. Sie gestattet durch weitere Pseudobefehle das Einfügen von Assemblerbedingungen in das Programm: IF (wenn), gefolgt von einem Ausdruck, dann bei Bedarf ELSE (sonst) und schließlich alles abgeschlossen durch ENDIF. Immer wenn der auf IF folgende Ausdruck wahr ist, dann werden die Befehle zwischen IF und ELSE (oder zwischen IF und ENDIF, wenn ELSE fehlt) in das Programm eingefügt. Ist der Ausdruck falsch, so wird entweder die Befehlskette zwischen ELSE und ENDIF eingefügt oder, falls kein ELSE vorliegt, gleich zu den auf ENDIF folgenden Befehle übergegangen.
Die Möglichkeit, Assemblerbedingungen vorzugeben, erlaubt dem Programmierer, ein Programm von vornherein für eine Vielzahl verschiedener Anwendungsfälle zu erstellen und dann je nach Einsatzbedingung nur die in Frage kommenden Segmente in das endgültige Objektprogramm einzubeziehen lassen. So kann man z.B. ein Programm zur Steuerung beliebig vieler Verkehrsampeln an einer Kreuzung im Vornherein erstellen. Später erhält man dann die genauen Anforderungen zur Zahl der zu steuernden Ampeln und zu den einzusetzenden Algorithmen. Der Programmierer

braucht dann nur noch ein paar Parameter im Quellentext setzen und das Programm unter Berücksichtigung dieser Bedingungen assemblieren zu lassen. Die bedingte Assemblierung ergibt dann ein auf den vorliegenden Anwendungsfall „maßgeschneidertes" Programm, das ohne viel Mehraufwand nur die benötigten Routinen enthält.
Bedingte Assemblierung ist daher vor allem im industriellen Bereich von besonderem Wert, wo auf allgemeine Lösungen viele Anwendungsfälle mit Sonderanforderungen existieren. Man kann dann – ist das Gesamtprogramm erst einmal fehlerfrei erstellt – sehr viel Entwicklungszeit einsparen, indem man bei jeder konkreten Anwendung nur noch ein paar Assemblierungsparameter vorgibt und den Rest automatisch ablaufen läßt.

Zusammenfassung

In diesem Kapitel wurden die Techniken und die Hard- und Softwarewerkzeuge vorgestellt, die man zur Programmerstellung benötigt, wobei die verschiedenen Möglichkeiten gegeneinander abgewogen worden sind.
Diese Möglichkeiten reichen auf Hardwareebene vom einfachen Einkartencomputer zum voll ausgebauten Entwicklungssystem. Auf Softwareebene wird der Bereich von rein binärer Kodierung bis hin zur Verwendung höherer Programmiersprachen erfaßt. Man muß unter diesen Möglichkeiten und Werkzeugen nach den eigenen Zielvorstellungen und dem Geldbeutel auswählen.

KAPITEL 11
SCHLUSSBEMERKUNGEN

Wir haben damit alle wichtigen Gesichtspunkte für die Programmierung behandelt, darunter die allgemeinen Definitionen und Grundkonzepte, die Handhabung der internen 6502-Register, die verschiedenen Möglichkeiten zur Speicheradressierung, die Verwaltung von Ein/Ausgabeeinheiten und die Kennzeichen der verschiedenen Programmentwicklungshilfen. Was kommt als nächstes? Wir können das unter zwei Gesichtspunkten betrachten. Der eine bezieht sich auf die technologische Weiterentwicklung, der zweite bezieht die Entwicklung Ihrer Kenntnisse und Fähigkeiten. Sehen wir uns diese beiden Punkte näher an.

Technologische Weiterentwicklung

Die Fortschritte in der MOS-Integrationstechnologie machen die Entwicklung immer komplexerer Chips möglich. Dabei sinken die Kosten zur Implementierung der Prozessorfunktionen ständig. Das bewirkt, daß viele der Ein/Ausgabechips ebenso wie die in einem System benötigten Peripheriesteuerbausteine einen einfachen Prozessor beinhalten. Insbesondere heißt das, daß die meisten LSI-Chips *programmierbar* werden. Damit entwickelt sich eine zusätzliche Schwierigkeit bei der Konzeption eines Systems. Um die Softwareaufgaben zu vereinfachen und gleichzeitig den Bausteinaufwand zu senken, enthalten die neueren E/A-Bausteine ausgefeilte Programmiermöglichkeiten: Viele Algorithmen, die vorher ausdrücklich im Anwendungsprogramm erarbeitet werden mußten, sind jetzt auf dem Chip integriert. Das hat jedoch die Nebenwirkung, daß die Programmerstellung durch die Tatsache erschwert wird, daß all diese E/A-Chips sich sehr stark voneinander unterscheiden und vom Programmierer vor dem Einsatz in allen Einzelheiten studiert werden müssen! *Programmieren eines Mikrocomputersystems bedeutet nicht mehr nur Programmieren des Prozessors selbst, sondern auch Programmieren all der Chips, die in das System eingebunden sind.* Dabei kann der zum Erlernen der Programmanforderungen der verschiedenen Chips erforderliche Aufwand ganz beachtlich werden.
Das ist natürlich nur eine Seite der Angelegenheit. Wenn diese Chips nicht verfügbar wären, so müßten die betreffenden Interfaces sehr viel komplexer sein, und der Aufwand für die mit ihnen zusammenarbeitenden Programme wäre noch größer. Diese Schwierigkeit wurde durch die neuen Anforderungen ersetzt, vor sinnvollem Einsatz eines der neuen Chips dessen Softwareanforderungen in allen Einzelheiten erst erlernen zu müssen. Es ist jedoch zu hoffen, daß die in diesem Buch vorgestellten Konzepte und Techniken das Vertrautmachen mit den zusätzlichen Möglichkeiten zu einer Angelegenheit mit vertretbarem Aufwand werden lassen.



Bild 11-1: Der PET: Ein typischer Heimcomputer mit allen Bestandteilen in einem Gehäuse

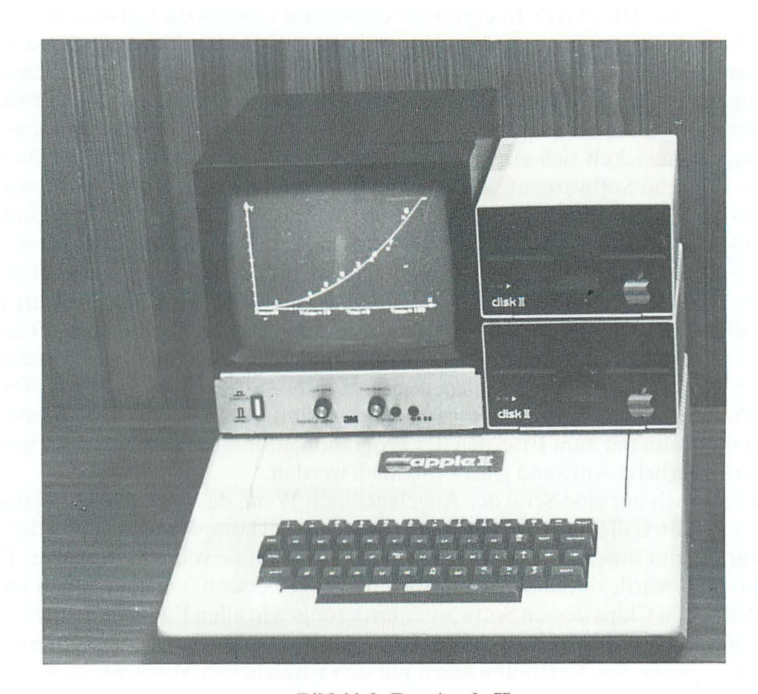


Bild 11-2: Der Apple II

Der nächste Schritt

Sie haben jetzt die Grundtechniken gelernt, mit denen einfache Programme auf dem Papier programmiert werden können. Das war das Ziel des Buchs hier. Der nächste Schritt ist nun, aus diesen Kenntnissen Fertigkeiten werden zu lassen. Sie müssen selbst programmieren. Es gibt keinen anderen Weg. Man kann das Programmieren auf gar keinen Fall nur auf dem Papier erlernen. Man braucht die unmittelbare Erfahrung mit dem, was man auf dem Papier erstellt hat, d.h. man muß den ganzen Zyklus von der ersten Programmkonzeption bis hin zur Fehlersuche durchgehen. Sie sollten nun in der Lage sein, diesen Schritt zu tun und Ihre eigenen Programme zu schreiben. Wenn Sie auf diesem Weg weitere Unterstützung möchten, so seien Ihnen die folgenden Titel empfohlen, die den hier vermittelten Stoff ausbauen und anwenden: Das Buch „6502 Anwendungen“ (SYBEX Nr. D302D) enthält eine Reihe praktischer Anwendungen, die bereits mit einfachen Einkartencomputern verwirklicht werden können. Dann gibt es das „6502 Advanced Programming“ (SYBEX Nr. G402A), das Programmiertechniken für komplexere Algorithmen behandelt. Außerdem ist noch ein in Microsoft BASIC geschriebener 6502-Assembler verfügbar.

ANHANG A

HEXADEZIMAL UMWANDLUNGSTABELLE

HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	00	000
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	256	4096
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	512	8192
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	768	12288
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	1024	16384
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	1280	20480
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	1536	24576
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	1792	28672
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	2048	32768
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	2304	36864
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	2560	40960
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	2816	45056
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	3072	49152
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	3328	53248
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	3584	57344
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	3840	61440

5		4		3		2		1		0	
HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ	HEX	DEZ
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393.216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

ANHANG B

6502 BEFEHLSSATZ – ALPHABETISCH

ADC	Addieren mit Übertrag	**JSR**	Verzweigen in Unterprogramm
AND	Logische UND	**LDA**	Laden Akkumulator
ASL	Arithmetisches Linksschieben	**LDX**	Laden X
BCC	Verzweigen, wenn Übertrag gelöscht	**LDY**	Laden Y
BCS	Verzweigen, wenn Übertrag gesetzt	**LSR**	Logisches Rechtsschieben
BEQ	Verzweigen, wenn Result = 0	**NOP**	Leerbefehl (keine Operation)
BIT	Teste Bit	**ORA**	Logisches ODER
BMI	Verzweigen, wenn Minus	**PHA**	Push A
BNE	Verzweigen, wenn ungleich 0	**PHP**	Push P Status
BPL	Verzweigen, wenn Plus	**PLA**	Pop A
BRK	Abbruch	**PLP**	Pop P Status
BVC	Verzweigen, wenn Überlauf gelöscht	**ROL**	Linksrotieren
BVS	Verzweigen, wenn Überlauf gesetzt	**ROR**	Rechtsrotieren
CLC	Übertrag löschen	**RTI**	Rückkehr von Unterbrechung
CLD	Dezimal-Flagge löschen	**RTS**	Rückkehr aus Unterprogramm
CLI	Unterbrechungsabschaltung löschen	**SBC**	Subtrahieren mit Übertrag
CLV	Überlauf löschen	**SEC**	Übertrag setzen
CMP	Vergleichen mit Akkumulator	**SED**	Dezimal setzen
CPX	Vergleichen mit X	**SEI**	Unterbrechungsabschaltung setzen
CPY	Vergleichen mit Y	**STA**	Akkumulator speichern
DEC	Dekrementieren Speicher	**STX**	X speichern
DEX	Dekrementieren X	**STY**	Y speichern
DEY	Dekrementieren Y	**TAX**	A nach X übertragen
EOR	Exklusives ODER	**TAY**	A nach Y übertragen
INC	Inkrementieren Speicher	**TSX**	SP (Stapelzeiger) nach X übertragen
INX	Inkrementieren X	**TXA**	X nach A übertragen
INY	Inkrementieren Y	**TXS**	X nach SP (Stapelzeiger) übertragen
JMP	Verzweigen	**TYA**	Y nach A übertragen

ANHANG C

6502 BEFEHLSSATZ – BINÄR

ADC	011bbb01	JSR	00100000
AND	001bbb01	LDA	101bbb01
ASL	000bbb10	LDX	101bbb10
BCC	10010000	LDY	101bbb00
BCS	10110000	LSR	010bbb10
BEQ	11110000	NOP	11101010
BIT	0010b100	ORA	000bbb01
BMI	00110000	PHA	01001000
BNE	11010000	PHP	00001000
BPL	00010000	PLA	01101000
BRK	00000000	PLP	00101000
BVC	01010000	ROL	001bbb10
BVS	01110000	ROR	011bbb10
CLC	00011000	RTI	01000000
CLD	11011000	RTS	01100000
CLI	01011000	SBC	111bbb01
CLV	10111000	SEC	00111000
CMP	110bbb01	SED	11111000
CPX	1110bb00	SEI	01111000
CPY	1100bb00	STA	100bbb01
DEC	110bb110	STX	100bb110
DEX	11001010	STY	100bb100
DEY	10001000	TAX	10101010
EOR	010bbb01	TAY	10101000
INC	111bb110	TSX	10111010
INX	11101000	TXA	10001010
INY	11001000	TXS	10011010
JMP	01b01100	TYA	10011000

Siehe Kapitel 4 für eine Definition des „bb“ Feldes.

ANHANG D

6502-BEFEHLSSATZ: HEXADEZIMAL MIT ZEITANGABEN

		Inplizit			Akkumulator			Absolut			Seite 0			Unmittelbar			Abs X		
Mnemonisch		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
A D C	(1)							6D	4	3	65	3	2	69	2	2	7D	4	3
A N D	(1)							2D	4	3	25	3	2	29	2	2	3D	4	3
A S L					OA	2	1	OE	6	3	06	5	2				1E	7	3
B C C	(2)																		
B C S	(2)																		
B E Q	(2)																		
B I T								2C	4	3	24	3	2						
B M I	(2)																		
B N E	(2)																		
B P L	(2)																		
B R K		00	7	1															
B V C	(2)																		
B V S	(2)																		
C L C		18	2	1															
C L D		D8	2	1															
C L I		58	2	1															
C L V		B8	2	1															
C M P								CD	4	3	C5	3	2	C9	2	2	DD	4	3
C P X								EC	4	3	E4	3	2	EO	2	2			
C P Y								CC	4	3	C4	3	2	CO	2	2			
D E C								CE	6	3	C6	5	2				DE	7	3
D E X		CA	2	1															
D E Y		88	2	1															
E O R	(1)							4D	4	3	45	3	2	49	2	2	5D	4	3
I N C								EE	6	3	E6	5	2				FE	7	3
I N X		E8	2	1															
I N Y		C8	2	1															
J M P								4C	3	3									
J S R								20	6	3									
L D A	(1)							AD	4	3	A5	3	2	A9	2	2	BD	4	3
L D X	(1)							AE	4	3	A6	3	2	A2	2	2			
L D Y	(1)							AC	4	3	A4	3	2	A0	2	2	BC	4	3
L S R					4A	2	1	4E	6	3	46	5	2				5E	7	3
N O P		EA	2	1															
O R A								0D	4	3	05	3	2	09	2	2	1D	4	3
P H A		48	3	1															
P H P		08	3	1															
P L A		68	4	1															
P L P		28	4	1															
R O L					2A	2	1	2E	6	3	26	5	2				3E	7	3
R O R					6A	2	1	6E	6	3	66	5	2				7E	7	3
R T I		40	6	1															
R T S		60	6	1															
S B C	(1)							ED	4	3	E5	3	2	E9	2	2	FD	4	3
S E C		38	2	1															
S E D		F8	2	1															
S E I		78	2	1															
S T A								8D	4	3	85	3	2				9D	5	3
S T X								8E	4	3	86	3	2						
S T Y								8C	4	3	84	3	2						
T A X		AA	2	1															
T A Y		A8	2	1															
T S X		BA	2	1															
T X A		8A	2	1															
T X S		9A	2	1															
T Y A		98	2	1															

(1) 1 zu n dazuzählen wenn Seitengrenze überschritten wird

Abs Y			(IND. X)			(IND)Y			Z Seite X			Relativ			Indirekt			Z Seite Y			Prozessor Status Kodes							
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N	V		B	D	I	Z	C
79	4	3	61	6	2	71	5	2	75	4	2										●	●					●	●
39	4	3	21	6	2	31	5	2	35	4	2										●						●	
									16	6	2										●						●	●
												90	2	2														
												B0	2	2														
												F0	2	2														
																					M_7	M_6					●	
												30	2	2														
												D0	2	2														
												10	2	2														
																								1		1		
												50	2	2														
												70	2	2														
																												0
																									0			
																										0		
																						0						
D9	4	3	C1	6	2	D1	5	2	D5	4	2										●						●	●
																					●						●	●
																					●						●	●
									D6	6	2										●						●	
																					●						●	
																					●						●	
59	4	3	41	6	2	51	5	2	55	4	2										●						●	
									F6	6	2										●						●	

																					●						●	
																					●						●	
															6C	5	3											
B9	4	3	A1	6	2	B1	5	2	B5	4	2										●						●	
BE	4	3																B6	4	2	●						●	
									B4	4	2										●						●	
									56	6	2										0						●	●
19	4	3	01	6	2	11	5	2	15	4	2										●						●	
																					●						●	
																					●	●	●	●	●	●	●	●
									36	6	2										●						●	●
									76	6	2										●						●	●
																					●	●	●	●	●	●	●	●
F9	4	3	E1	6	2	F1	5	2	F5	4	2										●	●					●	●
																												1
																									1			
																										1		
99	5	3	81	6	2	91	6	2	95	4	2																	
																		96	4	2								
									94	4	2																	
																					●						●	
																					●						●	
																					●						●	
																					●						●	
																					●						●	

(2) 2 zu n dazuzählen wenn in einer Seite gesprungen wird
3 zu n dazuzählen wenn in eine andere Seite gesprungen wird

ANHANG E

ASCII TABELLE

HEX LSD	MSD BITS	0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SPACE	0	@	P	–	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	–	=	M	]	m	}
E	1110	SO	RS	.	>	N	∧	n	~
F	1111	SI	US	/	?	O	←	o	DEL

DIE ASCII SYMBOLE

NUL	Null
SOH	Beginn des Kopfes (Start of Heading)
STX	Beginn des Textes (Start of Text)
ETX	Ende des Textes (End of Text)
EOT	Ende der Übertragung (End of Transmission)
ENQ	Anfrage (Enquing)
ACK	Bestätigung (Acknowledge)
BEL	Klingel (Bell)
BS	Zurücksetzen (Backspace)
HT	Horizontaler Tabulator (Horizontal Tabulation)
LF	Zeilenvorschub (Line Feed)
VT	Vertikaler Tabulator (Vertical Tabulation)
FF	Format Vorschub (Form Feed)
CR	Wagenrücklauf/Zeilenwechsel (Carriage Return)
SO	Rückschaltung (Shift Out)
SI	Dauerumschaltung (Shift In)
DLE	Datenverbindungs-Umschaltung (Data Link Escape)
DC	Gerätesteuerung (Device Control)
NAK	Negativ-Bestätigung (Negative Acknowledge)
SYN	Synchronisations-Leerlauf (Synchronous Idle)
ETB	Ende des Übertragungsblockes (End of Transmission Block)
CAN	Annullieren (Cancel)
EM	Datenträgerende (End of Medium)
SUB	Ersetzen (Substitute)
ESC	Umschaltung (Escape)
FS	Dateitrennzeichen (File Separator)
GS	Gruppentrennzeichen (Group Separator)
RS	Satztrennzeichen (Record Separator)
US	Einheiten-Trennzeichen (Unit Separator)
SP	Leerzeichen (Space/Blank)
DEL	Löschzeichen (Delete)

ANHANG F
RELATIVE SPRUNGTABELLE

RELATIVE SPRÜNGE VORWÄRTS

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127

RELATIVE SPRÜNGE RÜCKWÄRTS

MSD \ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	128	127	126	125	124	123	122	121	120	119	118	117	116	115	114	113
9	112	111	110	109	108	107	106	105	104	103	102	101	100	99	98	97
A	96	95	94	93	92	91	90	89	88	87	86	85	84	83	82	81
B	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65
C	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49
D	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
E	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
F	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

ANHANG G

HEXADEZIMALE BEFEHLSLISTE

LSD / MSD	0	1	2	3	4	5	6	7
0	BRK	ORA-I, X				ORA-Ø-P	ASL-Ø-P	
1	BPL	ORA-I, Y				ORA-Ø-P, X	ASL-Ø-P, X	
2	JSR	AND-I, X			BIT-Ø-P	AND-Ø-P	ROL-Ø-P	
3	BMI	AND-I, Y				AND-Ø-P, X	ROL-Ø-P, X	
4	RTI	EOR-I, X				EOR-Ø-P	LSR-Ø-P	
5	BVC	EOR-I, Y				EOR-Ø-P, X	LSR-Ø-P, X	
6	RTS	ADC-I, X				ADC-Ø-P	ROR-Ø-P	
7	BVS	ADC-I, Y				ADC-Ø-P, X		
8		STA-I, X			STY-Ø-P	STA-Ø-P	STX-Ø-P	
9	BCC	STA-I, Y			STY-Ø-P, X	STA-Ø-P, X	STX-Ø-P, Y	
A	LDY-IMM	LDA-I, X	LDX-IMM		LDY-Ø-P	LDA-Ø-P	LDX-Ø-P	
B	BCS	LDA-I, Y			LDY-Ø-P, X	LDA-Ø-P, X	LDX-Ø-P, Y	
C	CPY-IMM	CMP-I, X			CPY-Ø-P	CMP-Ø-P	DEC-Ø-P	
D	BNE	CMP-I, Y				CMP-Ø-P, X	DEC-Ø-P, X	
E	CPX-IMM	SBC-I, X			CPX-Ø-P	SBC-Ø-P	INC-Ø-P	
F	BEQ	SBC-I, Y				SBC-Ø-P, X	INC-Ø-P, X	

8	9	A	B	C	D	E	F	LSD / MSD
PHP	ORA-IMM	ASL-A			ORA	ASL		0
CLC	ORA, Y				ORA, X	ASL, X		1
PLP	AND-IMM	ROL-A		BIT	AND	ROL		2
SEC	AND, Y				AND, X	ROL, X		3
PHA	EOR-IMM	LSR-A		JMP	EOR	LSR		4
CLI	EOR, Y				EOR, X	LSR, X		5
PLA	ADC-IMM	ROR-A		JMP-I	ADC	ROR		6
SEI	ADC, Y				ADC, X			7
DEY		TXA		STY	STA	STX		8
TYA	STA, Y	TXS			STA, X			9
TAY	LDA IMM	TAX		LDY	LDA	LDX		A
CLV	LDA, Y	TSX		LDY, X	LDA, X	LDX, Y		B
INY	CMP-IMM	DEX		CPY	CMP	DEC		C
CLD	CMP, Y				CMP, X	DEC, X		D
INX	SBC-IMM	NOP		CPX	SBC	INC		E
SED	SBC, Y				SBC, X	INC, X		F

I = Indirekt
ØP = Seite 0

ANHANG H

DEZIMAL ZU BCD-UMWANDLUNG

DECIMAL	BCD	DEC	BCD	DEC	BCD
0	0000	10	00010000	90	10010000
1	0001	11	00010001	91	10010001
2	0010	12	00010010	92	10010010
3	0011	13	00010011	93	10010011
4	0100	14	00010100	94	10010100
5	0101	15	00010101	95	10010101
6	0110	16	00010110	96	10010110
7	0111	17	00010111	97	10010111
8	1000	18	00011000	98	10011000
9	1001	19	00011001	99	10011001

INHALTSREGISTER

Die SYBEX Bibliothek

MEIN ERSTER COMPUTER
von Rodnay Zaks – eine Einführung für alle die den Kauf oder die Nutzung eines Mikrocomputers erwägen. 305 Seiten, 150 Abbildungen, Format DIN A5, Ref.Nr.: **C200D**

CP/M HANDBUCH MIT MP/M
von Rodnay Zaks – ein umfassendes Lehr- und Nachschlagewerk für CP/M, das Standard Betriebssystem für Mikrocomputer. 310 Seiten, 100 Abbildungen, Format DIN A5, Ref.Nr.: **C300D**

MIKROPROZESSOR INTERFACE TECHNIKEN (3. überarbeitete Ausgabe)
von Rodnay Zaks/Austin Lesea – Hardware und Software Verbindungstechniken samt Digital/Analog-Wandler, Peripheriegeräte, Standard-Busse und Fehlersuchtechniken. 435 Seiten, 400 Abbildungen, Format DIN A5, Ref.Nr.: **C207D**

PROGRAMMIERUNG DES 6502 (2. überarbeitete Ausgabe)
von Rodnay Zaks – Programmierung in Maschinensprache mit dem Mikroprozessor 6502, von den Grundkonzepten bis hin zu fortgeschrittenen Informationsstrukturen. 350 Seiten, 160 Abbildungen, Format DIN A5, Ref.Nr.: **C202D**

PROGRAMMIERUNG DES Z80
von Rodnay Zaks – ein kompletter Lehrgang in der Programmierung des Z80 Mikroprozessors und eine gründliche Einführung in die Maschinensprache. 630 Seiten, 200 Abbildungen, Format DIN A5, Ref.Nr.: **C280D**

EINFÜHRUNG IN PASCAL UND UCSD/PASCAL
von Rodnay Zaks – das Buch für jeden der die Programmiersprache PASCAL lernen möchte. Vorkenntnisse in Computerprogrammierung werden nicht vorausgesetzt. Eine schrittweise Einführung mit vielen Übungen und Beispielen. 540 Seiten, 130 Abbildungen, Ref.Nr.: **P310D**

DAS PASCAL HANDBUCH
von Jacques Tiberghien – ein Wörterbuch mit jeder Pascal Anweisung und jedem Symbol, reservierten Wort, Bezeichner und Operator, für beinahe alle bekannten Pascal-Versionen. 510 Seiten, 270 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **P320D**

PASCAL PROGRAMME FÜR WISSENSCHAFTLER UND INGENIEURE
von Alan Miller – eine Sammlung von 60 der wichtigsten wissenschaftlichen Algorithmen samt Programmauflistung und Musterdurchlauf. Ein wichtiges Hilfsmittel für Pascal Benutzer mit technischen Anwendungen. 320 Seiten, 120 Abbildungen, Format 23 x 18 cm, Ref.Nr.: **P340D**

POCKET MIKROCOMPUTER LEXIKON
– die schnelle Informations-Börse! 1300 Definitionen, Kurzformeln, technische Daten, Lieferanten-Adressen und ein englisches Glossar. 140 Seiten, Format DIN A6, Ref.Nr. **X20D**

BASIC COMPUTER SPIELE/Band 1
herausgegeben von David H. Ahl – die besten Mikrocomputerspiele aus der Zeitschrift „Creative Computing" in deutscher Fassung mit Probelauf und Programmlisting. 224 Seiten, 59 Abbildungen, Ref.Nr.: **G40D**

BASIC COMPUTER SPIELE/Band 2
herausgegeben von David H. Ahl – 84 weitere Mikrocomputerspiele aus „Creative Computing". Alle in Microsoft-BASIC geschrieben mit Listing und Probelauf. 224 Seiten, 61 Abbildungen, Ref.Nr.: **G80D**

PROGRAMMING THE Z80
by Rodnay Zaks 620 pp., 200 illustr., Ref. C280
A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

PROGRAMMING THE Z8000
by Richard Mateosian 300 pp., 125 illustr., Ref. C281
How to program the Z8000 16-bit microprocessor. Includes a description of the architecture and function of the Z8000 and its family of support chips.

THE CP/M HANDBOOK (with MP/M)
by Rodnay Zaks 330 pp., 100 illustr., Ref. C300
An indispensable reference and guide to CP/M – the most widely used operating system for small computers.

INTRODUCTION TO PASCAL (Including UCSD PASCAL)
by Rodnay Zaks 420 pp., 130 illustr., Ref. P310
A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

THE PASCAL HANDBOOK
by Jacques Tiberghien 490 pp., 350 illustr., Ref. P320
A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS
by Alan Miller 400 pp., 80 illustr., Ref. P340
A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes such programs as curve-fitting, integrals and statistical techniques.

50 PASCAL PROGRAMS
by Rudolph Langer and Rodnay Zaks 275 pp., 90 illustr., Ref. P350
A collection of 50 Pascal programs ranging from mathematics to business and games programs. Explains programming techniques and provides actual pracitce.

APPLE PASCAL GAMES
by Douglas Hergert and Joseph T. Kalash 380 pp., 40 illustr., Ref. P360
A collection of the most popular computer games in Pascal challenging the reader not only to play but to investigate how games are implemented on the computer.

INTRODUCTION TO UCSD PASCAL SYSTEMS
by Charles T. Grant and Jon Butah 300 pp., 110 illustr., Ref. P370
A simple, clear introduction to the UCSD Pascal Operating System for beginners through experienced programmers.

INTERNATIONAL MICROCOMPUTER DICTIONARY
140 pp., Ref. X2
All the definitions and acronyms of microcomputer jargon defined in a handy pocket-size edition. Includes translations of the most popular terms into ten languages.

MICROPROGRAMMED APL IMPLEMENTATION
by Rodnay Zaks 350 pp., Ref. Z10
An expert-level text presenting the complete conceptual analysis and design of an APL interpreter, and actual listings of the microcode.

Fordern Sie ein Verzeichnis unserer Verlagsproduktion an.



SYBEX INC.
2344 Sixth Street
Berkeley,
California 94710, USA
Tel.: (415) 848-8233
Telex: 336311

SYBEX-EUROPE
4 Place Felix Eboué
75583 Paris Cedex 12
Tel.: 1/347-30-20
Telex: 211801

SYBEX-VERLAG
Heyestr. 22
4000 Düsseldorf 12
Tel.: (02 11) 28 70 66
Telex: 8 588 163

Die Presse über das vorliegende Buch:

„Der Stil ist klar und direkt. Der Inhalt ist gut organisiert. Wenn man die Anzahl der Bücher zum Thema Programmierung betrachtet, bekommt dieses eine hohe Punktzahl. Sie müssen sich keine sehr große Mühe geben, um den Inhalt zu begreifen. Er ist verständlich geschrieben. „EDN"

In den Programmen, die alle sehr sorgfältig entwickelt worden sind, werden alle Grundkonzepte erklärt und weiterentwickelt. Das Buch präsentiert gute Erklärungen. Der Autor hat für ein sehr breites Publikum geschrieben, vom Anfänger bis zum fortgeschrittenen Leser. „Elektronik"

Dieses Buch

wurde so geschrieben, daß Sie das Programmieren von Grund auf erlernen können. Da es eines der erfolgreichsten Bücher ist, die jemals im SYBEX-Verlag erschienen sind, wurde die 1. deutsche Ausgabe gründlich überarbeitet. Diese 2. Ausgabe enthält von Grund auf erneuerte Abbildungen, in denen alle Termini ins deutsche übersetzt wurden. Eine optimale Informationswiedergabe ist somit gewährleistet. Die verschiedenen Programmbeispiele und Techniken, die in diesem Buch vorgestellt werden, wenden sich an jeden Programmierer, gleich ob er Informationen zum 6502 Mikroprozessor sucht, Anfänger ist, oder schon Erfahrung im Programmieren hat.

Der Autor

hat Kurse in Programmierung über Mikroprozessoranwendungen vor mehreren Tausend Lernwilligen in der Welt gegeben. Er hat seinen Doktor in Computertechnologie von der kalifornischen Universität Berkeley. Er hat bereits einen programmierten APL-Anwendungskurs entwickelt und arbeitete in Silicon Valley in Anwendung und Entwicklung von industriellen Mikroprozessorsystemen, als diese das erste Mal auf den Markt kamen. Er hat einige der Bestseller im Bereich Microcomputerbücher auf den Markt gebracht, die jetzt bereits in 10 Sprachen verfügbar sind. Diese Buch, wie auch die anderen in dieser Serie, sind durch seine Erfahrung mit technischem und erzieherischem Hintergrund geschaffen worden.

ISBN: 3-88745-011-6